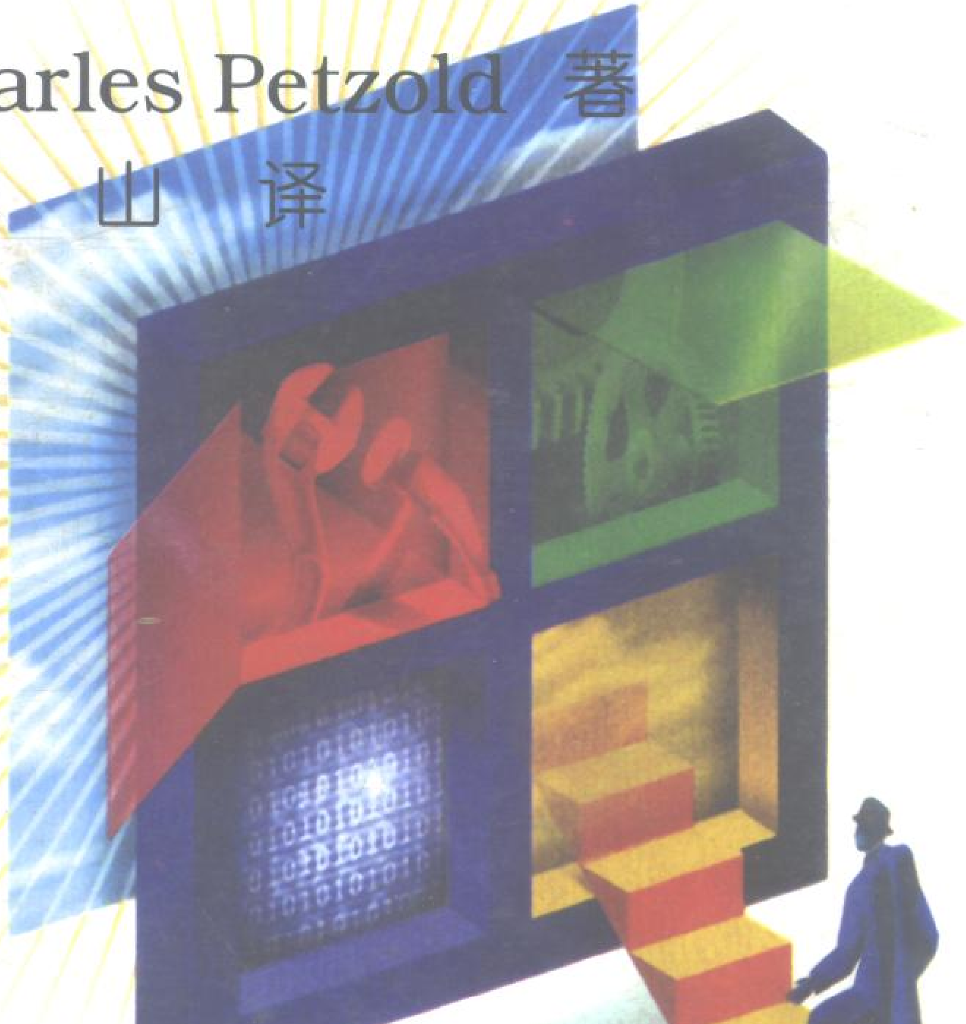
Microsoft Microsoft Microsoft

Windows® 95 程序设计

[美] Charles Petzold 著

郑全战 岚 山 译



清华大学出版社

Microsoft PRESS®

TP312
PZE/1

Windows 95 程序设计

Charles Petzold 著
郑全战 岚 山 译

藏书

清华大学出版社

0044519

(京)新登字 158 号

Windows 95 程序设计
Programming Windows 95
Charles Petzold

Copyright ©1996 by Charles Petzold

Original English language Edition Copyright ©1996 by Charles Petzold
Published by arrangement with the original publisher, Microsoft Press, a division of Microsoft Corporation, Redmond, Washington, U.S.A.

本书中文版由 Microsoft Press 授予清华大学出版社出版。

中华人民共和国国家版权局著作权合同登记章　图字：01-96-1448 号

版权所有，翻印必究。
本书封面贴有 Microsoft Press 激光防伪标签，无标签者不得销售。

JS527/07

书　　名：Windows 95 程序设计
作　　者：Charles Petzold　著　郑全战　岚山　译
出 版 者：清华大学出版社（北京清华大学校内，邮编 100084）
　　　　　因特网地址　www.tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京科技发行所
开　　本：787 × 1092 1/16　印张：55.75　字数：1321 千字
版　　次：1997 年 12 月第 1 版　1997 年 12 月第 1 次印刷
书　　号：ISBN 7-302-02619-X/TP・1343
印　　数：0001~5000
定　　价：94.80 元

目　　录

第一部分　开　　始

第 1 章　README.TXT....................3

1.1　对程序员的挑战....................3
1.2　基本原则....................4
1.3　Windows 简史....................6
1.4　本书简史....................7
1.5　让我们开始....................9

第 2 章　Hello, Windows 95....................10

2.1　WINDOWS 的特色....................10
2.1.1　图形用户界面(GUI)....................10
2.1.2　GUI 的概念和基本原理....................11
2.1.3　一致的用户界面....................11
2.1.4　多任务的优点....................11
2.1.5　内存管理....................12
2.1.6　与设备无关的图形界面....................12
2.1.7　投入 Windows....................13
2.1.8　函数调用....................13
2.1.9　面向对象的程序设计....................14
2.1.10　消息驱动式体系结构....................14
2.1.11　窗口过程....................14
2.2　编写第一个 Windows 程序....................15
2.2.1　传统程序的问题何在....................16
2.2.2　HELLOWIN 的文件....................16
2.2.3　Make 文件....................19
2.2.4　C 源代码文件....................20
2.2.5　Windows 函数调用....................20
2.2.6　大写字母标识符....................21
2.2.7　新的数据类型....................22
2.2.8　句柄简介....................22
2.2.9　匈牙利表示法....................23

2.2.10 程序的入口点......24
2.2.11 注册窗口类......25
2.2.12 创建窗口......28
2.2.13 显示窗口......29
2.2.14 消息循环......29
2.2.15 窗口过程......31
2.2.16 处理消息......32
2.2.17 播放声音文件......32
2.2.18 WM_PAINT 消息......33
2.2.19 WM_DESTROY 消息......34
2.3 Windows 程序设计的难点......34
2.3.1 别调用我，我会调用你......35
2.3.2 进队消息与不进队消息......36
2.3.3 别像只懒猪......37
2.3.4 学习曲线......37

第 3 章 输出文本......39

3.1 绘图和刷新......39
3.1.1 WM_PAINT 消息......40
3.1.2 有效矩形和无效矩形......40
3.2 GDI 简介......41
3.2.1 设备环境......41
3.2.2 获取设备环境句柄：方法一......42
3.2.3 绘图信息结构......43
3.2.4 获取设备环境句柄：方法二......44
3.2.5 TextOut：细节......45
3.2.6 系统字体......46
3.2.7 字符大小......47
3.2.8 文本尺度：细节......47
3.2.9 格式化文本......48
3.2.10 综合使用......49
3.2.11 SYSMETS1.C 窗口过程......54
3.2.12 空间不够......55
3.2.13 客户区的大小......55
3.3 滚动条......56
3.3.1 滚动条的范围和位置......57
3.3.2 滚动条消息......58
3.3.3 在 SYSTEMS 中加入滚动功能......59
3.3.4 绘图程序的组织......63

3.3.5 建立更好的滚动条....64
3.3.6 不用鼠标怎么办....69

第 4 章 图形基础....70

4.1 GDI 原理....70
4.2 GDI 的结构....71
4.2.1 函数调用的类型....72
4.2.2 GDI 图元....72
4.2.3 其它方面....73
4.3 设备环境....74
4.3.1 获取设备环境的句柄....74
4.3.2 获取设备环境信息....76
4.3.3 DEVCAPS1 程序....76
4.3.4 设备的大小....80
4.3.5 关于色彩....81
4.3.6 设备环境属性....82
4.3.7 保存设备环境....83
4.4 画线....84
4.4.1 边界框....89
4.4.2 Bezier 样条....95
4.4.3 使用现有画笔（stock pen）....100
4.4.4 画笔的创建、选择和删除....101
4.4.5 填充空隙....104
4.4.6 绘图方式....104
4.5 绘制填充区域....106
4.5.1 Polygon 函数和多边形填充方式....107
4.5.2 用画刷填充内部....108
4.6 映射方式....109
4.6.1 设备坐标和逻辑坐标....111
4.6.2 设备坐标系....111
4.6.3 视口和窗口....112
4.6.4 MM_TEXT 映射方式....113
4.6.5 “度量”映射方式....116
4.6.6 "自作主张的"映射方式....119
4.6.7 WHATSIZE 程序....124
4.7 矩形、区域和剪裁....127
4.7.1 矩形函数....127
4.7.2 随机矩形....129
4.7.3 创建和绘制区域....133

4.7.4 矩形与区域的剪裁......134
4.7.5 CLOVER 程序......135
4.8 路径......139
4.8.1 创建和绘制路径......139
4.8.2 扩展的画笔......140
4.9 BITS 与 BLTS......144
4.9.1 色彩和位图......145
4.9.2 设备无关位图（DIB)......145
4.9.3 DIB 文件......146
4.9.4 紧缩的 DIB 存储格式......147
4.9.5 显示 DIB......148
4.9.6 将 DIB 转换为位图对象......148
4.10 GDI 位图对象......149
4.10.1 在程序中创建位图......149
4.10.2 单色位图格式......150
4.10.3 彩色位图格式......152
4.10.4 内存设备环境......152
4.10.5 强大的 BLT......154
4.10.6 用 BitBlt 传输二进制位......157
4.10.7 DrawBitmap 函数......158
4.10.8 使用各种的 ROP 码......160
4.10.9 再论内存设备环境......162
4.10.10 色彩转换......165
4.10.11 映射方式转换......166
4.10.12 用 StretchBlt 放缩位图......166
4.10.13 画刷和位图......167
4.11 元文件......169
4.11.1 内存元文件的简单应用......169
4.11.2 将元文件保存在磁盘上......173
4.12 增强型元文件......173
4.12.1 改进......174
4.12.2 基本过程......174
4.12.3 内部机制......177
4.12.4 显示精确的图象......181
4.13 文本和字体......181
4.13.1 简单的文本输出......181
4.13.2 文本的设备环境属性......183
4.13.3 使用现有字体......185
4.13.4 字体类型......185

4.13.5 TrueType 字体......186
4.13.6 EZFONT 系统......187
4.13.7 内部机制......190
4.13.8 简单的文本格式化......191
4.13.9 处理段落......193

第二部分　输入手段

第 5 章　键盘......201

5.1 键盘基础......201
5.1.1 忽略键盘......201
5.1.2 焦点，焦点，谁获得了焦点......202
5.1.3 击键和字符......203
5.2 击键消息......203
5.2.1 系统击键与非系统击键......203
5.2.2 lParam 变量......204
5.2.3 虚拟键码......206
5.2.4 换档状态......208
5.2.5 使用击键消息......208
5.3 增强 SYSMETS：增加键盘接口......209
5.3.1 加入 WM_KEYDOWN 逻辑......209
5.3.2 发送消息......211
5.4 字符消息......216
5.4.1 WM_CHAR 消息......218
5.4.2 死字符消息......219
5.5 观察键盘消息......219
5.6 插入符（不是光标）......224
5.6.1 插入符函数......224
5.6.2 TYPER 程序......225
5.7 Windows 字符集......232
5.7.1 OEM 字符集......232
5.7.2 ANSI 字符集......233
5.7.3 OEM 、 ANSI 和字体......233
5.8 国际化考虑......234
5.8.1 字符集的处理......234
5.8.2 与 MS-DOS 交流......235
5.8.3 数字键盘的使用......236
5.8.4 Windows NT 的 Unicode 解决方案......237

第 6 章 鼠标 238

6.1 鼠标基础......238

6.1.1 一些简单的定义......239

6.2 客户区鼠标消息......239

6.2.1 简单的鼠标处理：一个例子......240

6.2.2 处理换档键......244

6.2.3 双击鼠标键......245

6.3 非客户区鼠标消息......246

6.3.1 命中测试......247

6.3.2 从消息产生消息......248

6.4 程序中的命中测试......249

6.4.1 一个假想的例子......249

6.4.2 示例程序......249

6.4.3 用键盘模拟鼠标......253

6.4.4 在 CHECKER 中加入键盘接口......254

6.4.5 将子窗口用于命中测试......258

6.4.6 CHECKER 中的子窗口......259

6.5 鼠标的捕获......263

6.5.1 画一个矩形......263

6.5.2 捕获的解决方案......267

6.5.3 BLOKOUT2 程序......268

第 7 章 定时器......272

7.1 定时器基础......273

7.1.1 系统和定时器......273

7.1.2 定时器消息不是异步的......273

7.2 定时器的使用：三种方法......274

7.2.1 方法一......274

7.2.2 方法二......280

7.2.3 方法三......283

7.3 定时器用于时钟......283

7.3.1 弹出式窗口的定位和缩放......287

7.3.2 获取日前和时间......287

7.3.3 国际化......288

7.3.4 构造模拟时钟......289

7.4 Windows 标准时间......294

7.5 动画......294

第 8 章 子窗口控制......298

8.1 按钮类 299
8.1.1 创建子窗口 303
8.1.2 子窗口向父窗口发消息 304
8.1.3 父窗口向子窗口发送消息 305
8.1.4 压入按钮 305
8.1.5 复选框 306
8.1.6 单选按钮 307
8.1.7 分组框 307
8.1.8 改变按钮文本 308
8.1.9 可见的和启用的按钮 308
8.1.10 按钮和输入焦点 309
8.2 控制与颜色 310
8.2.1 系统颜色 310
8.2.2 按钮颜色 311
8.2.3 WM_CTLCOLORBTN 消息 312
8.2.4 拥有者绘制按钮(Owner-Drawn Buttons) 313
8.3 静态类 319
8.4 滚动条类 320
8.4.1 COLORS1 程序 321
8.4.2 自动键盘接口 327
8.4.3 窗口子类化 328
8.4.4 给背景着色 328
8.4.5 给滚动条和静态文本着色 329
8.5 编辑类 330
8.5.1 编辑类风格 333
8.5.2 编辑控制通知 333
8.5.3 使用编辑控制 334
8.5.4 发送给编辑控制的消息 334
8.6 列表框类 335
8.6.1 列表框风格 336
8.6.2 将串放入列表框 337
8.6.3 选择和获取项 338
8.6.4 接收来自列表框的消息 339
8.6.5 一个简单的列表框应用程序 340
8.6.6 文件列表 343
8.6.7 Windows 的 Head 程序 344

第三部分　使用资源

第 9 章　图标、光标、位图和字符串..........351
9.1　编译资源..........351
9.2　图标和光标..........352
9.2.1　图象编辑器..........355
9.2.2　获取图标句柄..........357
9.2.3　在程序中使用图标..........358
9.2.4　使用其它的光标..........359
9.2.5　位图：象素组成的图象..........360
9.2.6　使用位图和画刷..........360
9.3　字符串..........364
9.3.1　使用字符串资源..........364
9.3.2　在消息框中使用字符串..........365
9.4　用户定义的资源..........366
第 10 章　菜单和加速键..........373
10.1　菜单..........373
10.1.1　菜单结构..........374
10.1.2　菜单模板..........374
10.1.3　在程序中引用菜单..........376
10.1.4　菜单和消息..........377
10.1.5　示例程序..........379
10.1.6　菜单设计规范..........385
10.1.7　定义菜单的复杂方法..........385
10.1.8　定义菜单的第三种方法..........386
10.1.9　浮动弹出式菜单..........387
10.1.10　使用系统菜单..........391
10.1.11　改变菜单..........394
10.1.12　其它菜单命令..........395
10.1.13　创建菜单的非正统方法..........396
10.2　在菜单中使用位图..........400
10.2.1　创建菜单位图的两种方法..........408
10.2.2　内存设备环境..........408
10.2.3　用文本创建位图..........409
10.2.4　位图的比例缩放..........410
10.2.5　将位图放入菜单中..........411
10.2.6　增加键盘接口..........413

10.3 加速键....413
10.3.1 为什么要使用加速键....413
10.3.2 安排加速键的几条规则....414
10.3.3 加速键表....414
10.3.4 加速键表的装入....416
10.3.5 键盘代码转换....416
10.3.6 接收加速键消息....417
10.3.7 菜单与加速键应用程序 POPPAD....418
10.3.8 启用菜单项....424
10.3.9 处理菜单项....424

第 11 章 对话框....427

11.1 模态对话框....427
11.1.1 创建一个"About"对话框....428
11.1.2 对话框模板....431
11.1.3 对话框过程....433
11.1.4 激活对话框....434
11.1.5 关于对话框风格的补充说明....435
11.1.6 关于定义控制的补充说明....436
11.1.7 一个更复杂的对话框....438
11.1.8 使用对话框控制....444
11.1.9 OK 和 Cancel 按钮....446
11.1.10 Tab 停留位和组....447
11.1.11 在对话框上画图....449
11.1.12 将其它函数用于对话框....450
11.1.13 定义自己的控制....450
11.2 消息框....457
11.2.1 弹出信息....458
11.3 非模态对话框....458
11.3.1 模态对话框与非模态对话框的区别....459
11.3.2 新的 COLORS 程序....461
11.3.3 HEXCALC：窗口还是对话框....466
11.3.4 创造性地使用控制 ID....472
11.4 通用对话框....472
11.4.1 增强 POPPAD....473
11.4.2 改变字体....492
11.4.3 查找与替换....492
11.4.4 只调用一个函数的 Windows 程序....493

第 12 章 现代用户界面....495

12.1 通用控制基础......496
12.1.1 库初始化......497
12.1.2 创建通用控制......498
12.1.3 通用控制风格......499
12.1.4 向通用控制发送消息......502
12.1.5 来自通用控制的通知消息......503
12.2 框架窗口控制......507
12.2.1 工具栏......507
12.2.2 创建工具栏......508
12.2.3 状态栏......524
12.2.4 GADGETS 程序......529
12.3 属性表格......561
12.3.1 创建属性表格......562
12.3.2 属性表格对话框过程......568
12.3.3 PROPERTY 程序......574

第四部分 核心和打印机

第 13 章 存储管理与文件 I/O......599

13.1 内存：好的、坏的和支离破碎的......599
13.1.1 分段内存......599
13.1.2 中期的解决方案......601
13.1.3 最后是无可争议的 32 位......602
13.2 内存分配......605
13.2.1 C 的库函数......605
13.2.2 Windows 95 内存分配基础......605
13.2.3 可移动的内存......606
13.2.4 可丢弃内存......608
13.2.5 其它函数和标志......608
13.2.6 指针合法吗？......608
13.2.7 虚拟内存函数......609
13.2.8 堆函数......610
13.3 文件输入/输出......610
13.3.1 以前的方式......610
13.3.2 Windows 95 的不同之处......610
13.3.3 Windows 95 文件 I/O 函数......611
13.3.4 内存映射 I/O......611

第 14 章 多任务和多线程......613

14.1　多任务的各种模式 613
14.1.1　DOS 下的多任务？ 613
14.1.2　非抢占式的多任务 614
14.1.3　PM 和串行化的消息队列 615
14.1.4　多线程解决方案 615
14.1.5　多线程体系结构 616
14.1.6　线程间的“争吵” 616
14.1.7　Windows 的好处 617
14.1.8　新的！先进的！有线程了！ 618
14.2　Windows 95 多线程 618
14.2.1　再论随机矩形 618
14.2.2　程序设计竞赛的问题 621
14.2.3　多线程解决方案 628
14.2.4　有问题吗？ 636
14.2.5　Sleep 的好处 637
14.3　线程同步 637
14.3.1　临界区 637
14.3.2　互斥(mutex)对象 639
14.4　事件信号 639
14.4.1　BIGJOB1 程序 639
14.4.2　事件对象 644
14.5　线程局部存储（TLS） 648

第 15 章　使用打印机 650

15.1　打印、后台处理和打印函数 651
15.2　打印机设备环境 654
15.2.1　获取 CreateDC 参数 654
15.2.2　修改后的 DEVCAPS1 程序 658
15.2.3　PrintProperties 调用 667
15.2.4　检查 BitBlt 支持 668
15.3　FORMFEED 程序 668
15.4　打印图形和文本 670
15.4.1　打印的基本过程 673
15.4.2　使用放弃过程来取消打印 675
15.4.3　Windows 如何使用 AbortProc 677
15.4.4　实现放弃过程 677
15.4.5　增加打印对话框 680
15.4.6　为 POPPAD 增加打印功能 684
15.4.7　处理错误码 689

15.5 打印分带技术......691
15.5.1 分带原理......691
15.5.2 分带技术的实现......694
15.6 打印机和字体......697

第五部分 数据交换和链接

第 16 章 剪贴板......701
16.1 剪贴板的简单使用......701
16.1.1 标准剪贴板数据格式......701
16.1.2 传送文本到剪贴板......702
16.1.3 从剪贴板上获取文本......703
16.1.4 打开和关闭剪贴板......704
16.1.5 使用位图剪贴板......705
16.1.6 元文件和元文件图片......706
16.2 复杂的剪贴板用法......710
16.2.1 利用多个数据项......710
16.2.2 延迟生成......711
16.2.3 私有数据格式......713
16.3 实现剪贴板浏览器......715
16.3.1 剪贴板浏览器链......715
16.3.2 剪贴板浏览器的函数和消息......715
16.3.3 一个简单的剪贴板浏览器......718
第 17 章 动态数据交换(DDE)......721
17.1 基本概念......722
17.1.1 应用程序、主题和项......722
17.1.2 对话的类型......723
17.1.3 字符串和原子......727
17.2 一个 DDE 服务器程序......728
17.2.1 DDEPOP1 程序......742
17.2.2 WM_DDE_INITIATE 消息......743
17.2.3 ServerProc 窗口过程......743
17.2.4 WM_DDE_REQUEST 消息......744
17.2.5 DDEPOP1 的 PostDataMessage 函数......745
17.2.6 WM_DDE_ADVISE 消息......746
17.2.7 更新数据项......747
17.2.8 WM_DDE_UNADVISE 消息......748
17.2.9 WM_DDE_TERMINATE 消息......748

17.3　一个 DDE 客户程序......748
17.3.1　发动 DDE 对话......756
17.3.2　WM_DDE_DATA 消息......757
17.3.3　WM_DDE_TERMINATE 消息......758
17.4　DDE 管理库......758
17.4.1　概念上的差异......758
17.4.2　DDE 的 DDEML 方式......759

第 18 章　多文档界面......771

18.1　MDI 的组成......771
18.2　WINDOWS 95 和 MDI......772
18.3　示例程序......773
18.3.1　三个菜单......785
18.3.2　程序初始化......785
18.4　创建子窗口......787
18.5　关于框架窗口的消息处理......787
18.6　子文档窗口......788
18.7　结束处理......790
18.8　窗口过程的强大功能......790

第 19 章　动态链接库......791

19.1　库的基本知识......791
19.1.1　库：一词多义......792
19.2　一个简单的 DLL......792
19.3　DLL 中的共享内存......796
19.3.1　STRLIB 库......797
19.3.2　库入口/出口点......800
19.3.3　STRPROG 程序......801
19.3.4　运行 STRPROG......806
19.3.5　在 STRPROG 实例之间共享数据......807
19.4　对库的一些限制......807
19.5　不用输入的动态链接......808
19.6　纯资源库......809

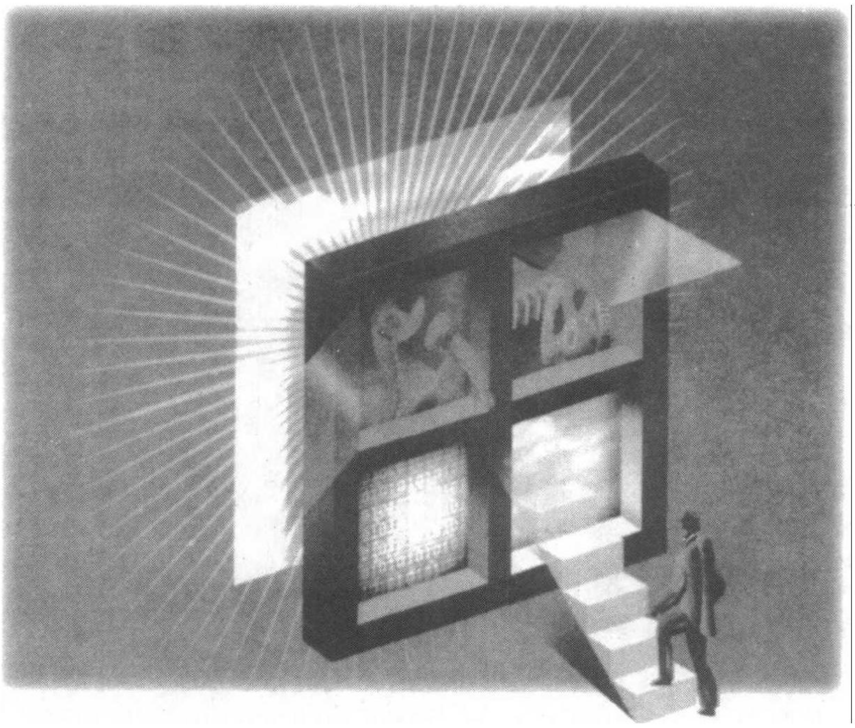
第 20 章　何谓 OLE......814

20.1　OLE 基础......815
20.1.1　连接到 OLE 库......815
20.1.2　解释结果代码......816
20.1.3　组件对象模型（COM）接口......819
20.1.4　IUnknown 服务......824

20.1.5 OLE 是一种客户/服务器规范吗？ 828
20.2 私有组件服务器........828
20.2.1 IMALLOC.DLL........833
20.2.2 那些古怪的宏 834
20.2.3 由 IUnknown 提供的服务........836
20.3 私有组件的客户 840
20.4 公共组件的服务器........849
20.4.1 登记表的作用........857
20.4.2 产生和使用 CLSID 的技术 860
20.4.3 类工厂组件........861
20.4.4 控制服务器的生命周期........863
20.5 公共组件客户 865
20.6 最后的话........874

第一部分

开始



第 1 章　README.TXT

这本书针对的是熟练掌握 C 编程语言并希望学习如何为 Microsoft Windows 95 操作系统编写应用程序的人们。熟悉 C 是使用这本书的三个前提之一，第二个前提是你安装了 Microsoft Visual C++版本 4.0 的 32 位开发环境，第三个前提是你确实使用过 Windows 95 并且对其用户界面的构成有一个基本了解。

你可能知道，Windows 95 是 Windows 图形操作系统的最新版本。Windows 系统最早是在 1985 年 11 月推出的，用于 IBM PC 及其兼容机。从市场占有来看，Windows 几乎完全击倒了过去十年中的所有竞争对手，成为个人计算机环境中的事实标准。现在，如果你正在为 PC 兼容机写程序的话，那么你肯定也是在为 Windows 环境写程序。

就把这一章当成你的第一课吧。尽管某些具有虐待倾向的教师们总是立即开始教材中枯燥难懂的部分，但是我们大多数人还是喜欢比较轻松的开场白。因此，在本章中，我将讨论 Windows 的一些历史背景，这本书的基本原则，以及一点关于我自己和我为什么编写这本书的原因。

不过，我不能保证本章对你会完全没有一点压力。你是一个程序员，你是一个软件工程师，和所有其它工程师一样，你的角色是承担起困难（具有挑战性——常常也是令人满意的）工作，从而使世界成为一个更轻松的世界。你是在修路、修桥，使人们更接近他们的目的地，这些东西必须是坚固的，耐用的，并且是高速的。

1.1　对程序员的挑战

在电影《大峡谷》中，一个父亲在帮助他 15 岁的儿子学习驾驶，他说“在洛杉矶进行左转是你在你的生命中学习的一件最艰难的事。”他也可以这样形容基于 Windows 的编程。

我并不是特指基于 Windows 的编程技术。我会在下一章中开始这个艰难的话题。现在我考虑更多的是程序设计原理，这是使用户更轻松的关键所在，也是一个相对比较新的概念。

计算机发展的早期，仅有的用户同时也是程序员。老式的计算机程序吃进去一些数据，嚼一会儿，然后再吐出一些纸来。根本没有计算机外行要用计算机这样的问题。即使当计算机软件变得有了一些交互性，能够在一个电传或者视频显示器上显示一个命令行，用户仍然需要记住密码似的命令和选项。

程序设计的真正革命应该是随着早期的字处理器（例如，WordStar）和电子表格（VisiCalc）的出现而到来的，这些应用软件实现了现代用户界面设计中最基本元素的初始形式。当然，这就是菜单。这些早期交互应用系统的菜单实现得并不怎么好，但是新的

思想已经出现，并且慢慢地发展成熟。回想起来，采用菜单似乎是天经地义的：它显示了用户可选的所有项目。当然，在早期，内存的缺少限制了程序员设计一种好的用户界面。用户不友好的程序简短并更容易编写；友好的程序更大并更难于编写。

“没有免费午餐这样的好事”的原则在这里是有效的。有些人不得不做艰难的工作。作为一个程序员，你就是这样一个人。

幸运的是，这项艰难工作的某些部分已经由 Microsoft Windows 95 操作系统的设计人员和程序员为你做了。这些无名英雄实现了创建最新用户界面对象以及使用丰富的格式化文本和图形显示程序输出所需的大部分代码。为此，他们也不得不创建了一个庞大的应用编程界面（API）。对于首次接触 Windows 的程序员来说，API 是相当可怕的。并非只有 Windows 才这样——所有的现代图形用户界面都存在这种问题。

从前，一般的估计是一个程序员需要大约 6 个月的时间才能学会 Windows 的编程（我经常开玩笑说，使用这本书，这件事可以被减少到 26 个星期，或者可能甚至只有 180 天）。最近几年，Windows 系统越来越庞杂，但同时也出现了可用的工具来帮助你解决其中的难点，因此 6 个月的原则仍然是适用的。

我告诉你这个是因为我不想让你由于不能立刻“获得”这门技术而觉得自己在技术上不行或者智力有缺陷。如果你不熟悉针对图形环境的编程，那么 Windows 对你来说可能非常陌生。不过，最终你会掌握它的。

1.2 基本原则

在这本书中，我将教给你的是我认为是“经典的” Windows 编程。我使用普通的旧式 C 编程语言（非 C++），并且我直接使用原始的应用编程界面（API），而没有使用将 API 隐藏在更简单界面下的任何“包装”。虽然 Windows 95 比 Windows 的早期版本又有了提高，但是这本书中的大多数程序看起来和在大约 10 年以前的 Microsoft Windows 版本 1.0 中编写的程序并没有太多的差别。

从一种意义上说，这本书走了一条 Windows 编程的艰难路径——不过也是最基础、最本质、用途最广和最强有力的方式。你没法用其它方法来让 Windows 做更多的事。而且，通过学习使用 C 和原始 API 的典型 Windows 编程，你可以更清楚地理解 Windows 是如何与你的应用进行交互的，而这通常是非常有用的知识。这种方法建立了一种坚实的基础，你将永远不会感到遗憾，相信我。

虽然我教的是经典的 Windows 编程，但是我并不是建议你将自己局限于此。在当今的日子里，你有许多选择可以使 Windows 的编程更加容易一些。一个流行的选择是 C++，通常和 Microsoft Foundation Classes(MFC)或者 Borland 的 Object Windows Library(OWL)等类库结合使用。其它选择是 Microsoft 的 Visual Basic 和 Borland 的 Delphi 。代码生成器也可以用来将大量的工作从 Windows 编程中分离出去。你甚至可以使用简单的，诸如 Asymetrix 的 ToolBook 等脚本语言来创建 Windows 应用。

我无法告诉你哪种选择是最好的；它确实依赖于你正在编写的应用，以及你愿意作多大的牺牲，将你自己和环境隔离开。

在这本书中我不会涉及的另一个话题是交互开发环境（ IDE ）——诸如 Microsoft Visual C++版本 4.0 等——的使用。这些环境通常通过在资源（诸如菜单和对话框等）创建方面提供帮助，或者通过提供给你编写、编译、运行和调试代码的通用环境来帮助你。我对 IDE 没有任何抵触。我认为它们非常伟大。但是，它们的作用经常会妨碍而不是帮助我学习经典的 Windows 编程。简而言之，我不想教会你学习如何填充 IDE 对话框；而是想教会你如何设计你自己的对话框和处理其中的输入。

由于这个原因，这本书中的代码看起来好像是在一个普通的文本编辑器中创建的，而程序看起来好像是从一个老式的命令行上进行编译的。当我们开始资源文件的话题时，这些现象将最明显，资源文件中包含对程序中的菜单、对话框和其它资源的描述。这本书中的资源文件是通过一个 IDE 产生的。我这么做是因为我希望让你明白在一个资源文件中的语句的含义。当通过一个 IDE 来生成 Make 文件——包含编译并链接你的程序以创建一个可执行文件的命令的文件——时，这个文件是出名的又长又复杂。我的文件却既短又简单，对此我一点也不感惭愧！在这本书中唯一的非文本资源是图标、鼠标光标和位图图象，自然它们通常需要使用工具来创建。

这种方法的好处在于你可以简单地将源文件拷贝到你的硬盘中，激活一个 MS-DOS 命令行窗口，运行 make 文件创建一个可执行文件，然后从命令行就可以运行可执行文件。

这本书中的大多数程序都很短，目的是演示一两个重要的概念，否则它们将没有显示的必要。这些程序是教学工具而不是完整的 Windows 程序的样板。例如，每个真正的 Windows 程序都应该有一个与众不同的图标；但我的程序却几乎没有这样做，因为这只会带来混乱。而且，通常还建议把一个程序使用的文本串放到一个资源文件中，以便于被翻译为不同的语言。但是，这样做也可能会引入混乱并干扰我试图强调的重点。我的经验是程序越小，一个读者就越可能会仔细地进行阅读。

这本书不可能覆盖 Windows 95 编程的所有方方面面，也不可能替代正式的文档。当吸收了这本书中的内容之后，你仍然需要花费许多时间来阅读 Windows 95 功能和函数调用的有关文档。

这不是微软出版社出版的仅有的 Windows 编程方面的书籍。关于基于 API 的 Windows 的C语言编程，另一种选择是由 Jeff Prosise 著的 Programming Windows 95 with MFC （1996 年夏出版）。为了更深入了解 Windows 95 的内部机制，可以看一看由 Adrian King 著的 Inside Windows 95 。第 12 章关于高级用户界面选项的讨论在由 Nancy Winnick Cluts 著的 Programming the Windows 95 User Interface 中有更详尽的阐释。第 20 章关于 OLE 的简短介绍应该由 Kraig Brockschmidt 著的 Inside OLE 来补充。其它有关 Windows 95 编程的话题可以在 Programmer's Guide to Microsoft Windows 95 中找到。另外还有 David J. Kruglinski 著的 Inside Visual C++，Adam Denning 著的 OLE Controls Inside Out 和 Bruce McKinney 著的 Hardcore Visual Basic 等，如果有兴趣，都可以读一读。

我编写（以及你阅读）这本书的最后一个基本原则是：我不会涉及 Windows 95 编程中“未公开的”或者说“未授权的”方面。虽然这类信息经常可能是有趣的，但是我不认为它们很重要，而只不过有些不寻常而已。当我成为一个熟练的程序员时，我越来越多地试着不要依赖于实现细节，因为这些细节可能会在操作系统的未来版本中被改变。当然，情况并不总是这样，并且经常会有一些未能被文档提到的“漏洞”。错误是可能存在的。

我不会忽略这些问题。

1.3 Windows 简史

在 1981 年秋天 IBM PC 推出之后不久，就已经很明显， PC 上的主流操作系统将是 MS-DOS 。 MS-DOS 代表 Microsoft Disk Operating System （磁盘操作系统）。 MS-DOS 的早期版本提供给用户一种命令行界面，以诸如 DIR 和 TYPE 这样的命令为特征。它可以将应用程序装入内存来执行，并且提供给这些程序访问文件，读键盘，以及写到视频显示器（只能以字符模式）和打印口等服务。

由于内存和硬件的限制，成熟的图形环境慢慢才到来。当 Apple Computers 不幸的 Lisa 在 1983 年 1 月发布时，它提供了另一种可能的选择，并在 1984 年 1 月成为了 Micintosh 上图形环境的一种标准，而且（尽管它的市场占有率在下降）至今仍然被认为是衡量所有其它图形环境的标准。

Windows 是由微软在 1983 年 11 月（在 Lisa 之后 Micintosh 之前）宣布，并在两年后（ 1985 年 11 月）发布。在此后的两年中，紧随着 Microsoft Windows 早期版本 1.0 之后，又推出了几种改进版本，以支持国际商业市场，并提供新型视频显示器和打印机的驱动程序。

Windows 版本 2.0 是在 1987 年 11 月正式投放市场的。该版本对用户界面作了一些改进。这些改进中最有效的是使用了重叠式窗口，而 Windows1.x 中使用的是“并列”窗口。 Windows 2.0 还增强了键盘和鼠标接口，特别是加入了菜单和对话框。

至此， Windows 还只要求一个 Intel 8086 或者 8088 微处理器，以“实模式”运行，访问 1MB 的内存。 Windows/386（在 Windows 2.0 之后不久发布的）使用 Intel 80386 微处理器的“虚拟 86 ”模式，实现直接访问硬件的多个 MS-DOS 程序的窗口化和多任务化。为了统一起见， Windows 版本 2.1 被更名为 Windows/286 。

Windows 版本 3.0 是在 1990 年 5 月 22 日举办的一次盛大的产品发布会上推出的。它将 Windows/286 和 Windows/386 结合到一种产品中。 Windows 3.0 有了一个很大的改变，这就是对 Intel 80286, 80386, 80486 微处理器保护模式的支持。这能使 Windows 和 Windows 应用程序访问高达 16MB 的内存。 Windows 用于运行程序和维护文件的“外壳”程序得到了全面的改进。 Windows 3.0 是第一个进入许多用户家里和办公室的版本。

Windows 版本 3.1 是在 1992 年 4 月推出的，其中包括的几个重要特性是 TrueType 字体技术（给 Windows 带来可缩放的轮廓字体）、多媒体（声音和音乐）、 OLE 和通用对话框。同样， Windows 3.1 只能在保护模式下运行，并且要求至少配置了 1MB 内存的 80286 或者 80386 微处理器。

任何 Windows 的历史都必须包括 OS/2 的一点介绍，这是对 DOS 和 Windows 的另一种选择，最初是由 Microsoft 和 IBM 合作开发的。 OS/2 版本 1.0 （只有字符模式）在 Intel 80286 （或者后来的）微处理器上运行，在 1988 年末发布。在 1988 年 10 月的 OS/2 版本 1.1 中出现了图形的表现管理器（ PM ）。 PM 最初的设想是成为 Windows 的一种保护模式版本，但是图形 API 改变程度太大，以致使软件生产商很难提供对这两种平台的支持。

直到在 1990 年 9 月，IBM 和 Microsoft 之间的冲突达到了顶峰，导致了这两个公司最终的分道扬镳。IBM 接管了 OS/2，而 Microsoft 明确表示 Windows 将是它操作系统策略的中心。虽然 OS/2 仍然拥有一些狂热的崇拜者，但是它远未达到 Windows 这样的普及程度。

在 1993 年 1 月引入的 Windows NT 是 Windows 支持 Intel 80386，80486 和 Pentium 微处理器的 32 位编程模式的第一个版本。Windows NT 有 32 位的平面地址空间和 32 位的整数。Windows NT 还可以被移植到基于几种 RISC 芯片的工作站上运行。

Windows 95（以前代号为 Chicago，有时还被称为 Windows 版本 4.0）是在 1995 年 8 月发布的。和 Windows NT 一样，Windows 95 也支持一种 32 位的编程模式（因此要求一个 80386 或者更高级的微处理器）。虽然它缺少 Windows NT 中的某些功能特性，诸如高安全性和对 RISC 机器的可移植性等，但是 Windows 95 具有可以在只有 4MB 内存的机器上运行的优点。

显然，为 Windows NT 和 Windows 95 以前的 16 位版本的 Windows 编写的程序通常不能透明地被移植到更新的 32 位版本的 Windows 中。我将在后面的章节中讨论一些移植所必要的修改。

Microsoft 试图通过定义 API 集来将它的不同平台区分开。Win16 API 由 Windows 3.1 支持，而 Win32 API 由 Windows NT 和 Windows 95 支持。但是等一等——还有更多。Microsoft 使程序员可以为 Windows 3.1 编写 32 位应用系统，其中 32 位的函数调用通过一个动态链接库（DLL）被翻译为 16 位的调用。API 被命名为 Win32s（“s”代表“子集”，因为这种 API 只支持在 Win16 中的函数），而且，有时 Windows 95 API 被命名为 Win32c（“c”代表“兼容的”），但是这个术语现在被抛弃了。

目前，Windows 95 和 Windows NT 都被认为能够支持 Win32 API。不过，每一种操作系统都支持另一种不支持的某些功能特性。它们的相同之处是相当可观的，因此有可能编写可以在两种系统上运行的程序。而且，人们普遍认为二者最终会合二为一。

1.4 本书简史

在 1988 年初，Programming Windows 是出现在书店书架上有关 Windows 编程的第一本书的第一版。当我为 Windows 95 准备修订 Programming Windows 时，我很震惊地意识到 1995 年是 Windows 的 10 周年，从而也是这本书诞生的 10 周年。这个事实既让人兴奋，又有那么一点点使人害怕。

当我回顾这本书的历史时，我认为这在很大程度上归因于偶然的机遇以及我和计算机领域中许多杰出人物的接触。事情的开始是这样的。

在 1985 年春，我在为 PC Magazine 作一些自由写作，并经常逗留在他们纽约 One Park 大街的办公楼。Microsoft 的 Steve Ballmer（现在是销售及支持的执行副总裁）偶尔会来到这里，带来人们翘首以待的叫做 Windows 的产品的最新测试版本。我们中有兴趣的家伙会聚集在高级编辑 John Dickinson 的办公室里来玩一玩这东西。在 PC Magazine 的职员中 John 是那些在他们的机器上有 EGA 的少数几个人之一，因此他可以在彩色方式下运行

Windows 。我们会玩一段 Windows 的这些早期版本（通常直到我们厌倦了它的崩溃为止），然后等待 Ballmer 下次来会带有一个更可靠的版本。

在 1985 年春的一天，我问 John Dickinson 应该如何实际地编写一个在 Windows 环境下运行的程序。 John 很快打开一个桌子的抽屉，将 Ballmer 几个星期以前留给他的几英寸厚的复印纸和大约 10 个磁盘都给了我。 John 的唯一要求是请我不要再把这些废物带回他的办公室。

这堆纸和磁盘是第一个 Windows Software Development Kit(SDK)的早期版本。我将这捆东西带回家，将 SDK 安装在一个 Bernoulli 盒子上（我当时使用一个带两个软驱的 IBM PC 进行工作），并且在经过了大约 6 个月的完全迷惑之后，立即成了一个 Windows 程序员。经过这一段时间的学习经历以及对文档的研究，我可以这样说 ：“我可以比 Microsoft 自己还要好得多地解释这些资料。”

Windows 版本 1.0 最终是在 1985 年 11 月发布的，但是当时还没有人预见到 Windows 将会在某一天统治市场。当然，它有一些竞争对手，包括 IBM 的 TopView ， Digital Research 的 GEM ，以及 Quarterdeck 的 DESQview 。在 1985 年 2 月 25 日， PC Magazine 的封面发表了“ Windows War ”（ Windows 战争）的宣言。这是我为这个杂志写的第一篇封面文章，虽然当时我是被分派去评论 TopView 而不是 Windows 。

又过了一年的时间，我才有足够的自信正式发表有关 Windows 编程的文章，这是在 Microsoft Systems Journal 的 1986 年 12 月刊上。我相信这篇文章——这是你将在第四章中看到的我的 WHATSIZE 程序的一个早期版本——是出现在一本杂志上的第一篇有关 Windows 编程的文章。我碰巧认识 MSJ 的首任编辑 Jonathan Lazarus ，因为他以前是发行 PC Magazine 的 Ziff-Davis 的一个副总裁。后来 Jon 去了 Microsoft 工作，目前他是那里的策略关系部门的副总裁。

为 MSJ 撰写有关 Windows 的文章使我得到了 Microsoft Press 的注意，但这也是一条迂回的道路。 1986 年 10 月，在华盛顿 Redmond 的一次 Microsoft 语言会议上遇到了 Tandy Trower （目前是 Microsoft 负责用户界面设计的主任），我告诉他我是多么荣幸将为 MSJ 撰写一篇有关 Windows 编程的文章。他将我的名字给了 Susan Lammers ，那时她是 Microsoft Press 的总编。 Microsoft Press 的人已经知道了我的名字，因为我曾经参与名声不好的 MS-DOS Encyclopedia （百科全书）第一个版本的校阅，并告诉他们其中包括的许多缺陷和错误，导致该版本撤出市场，并最终被完全推翻后在 Ray Duncan 的监督下重新来过。

1986年11月在洛杉矶的秋季Comdex期间，我遇到了Microsoft Press的新编辑Claudette Moore （现在是麻萨诸塞州的一个计算机著作代理商），并在一起苦心设计出了一个草图。最初， Programming Windows 应该是一本更薄的书，针对的读者是程序员和高级用户。但当我在下一年中写这本书时（在其间我不得不将重点由 Windows 1.0 转移到 Windows 2.0 ），它在尺寸上增大了，而在范围上却缩小了。

现在在你的手上的是 Programming Windows 的第四版本。我曾经针对 Windows 3.0 ，然后又针对 Windows 3.1 对它进行过修订。自从这本书第一次发行之后，就有许多 Windows 程序员告诉我：“ Programming Windows 将他们领入了一个非常陌生的环境”。没有什么能比这更让我高兴了。我不求名垂千史，只求能对 Windows 的成功有所帮助。

1.5 让我们开始

好了，第一天的课程即将结束。在下一章中，我们将开始接触一些 Windows 代码。为了做到这一点，你需要安装 Microsoft Visual C++版本 4.0 。在你的 AUTOEXEC.BAT 文件中，你需要包括下列语句：

```
CALL \MSDEV\BIN\VCVAR32.BAT
```

这是包含在 Visual C++中的一个文件，为从 MS-DOS 命令行编译程序设置 DOS 环境变量。它简单地指示头文件、库文件和二进制文件的路径。你还需要在你的 AUTOEXEC.BAT 文件中使用另一个 CALL 语句来运行 MSC.BAT 文件，如图 1-1 中所示。它还定义将在以后章节中的 make 文件中使用的环境变量。

MSC.BAT

```
REM -------------------------------------------------------------
REM  MSC.BAT -- Set up environment for Microsoft C/C++ 7.0 NMAKE
REM -------------------------------------------------------------
SET CC=cl
SET CFLAGS=-c -DSTRICT -G3 -Ow -W3 -Zp -Tp
SET CFLAGSMT=-c -DSTRICT -G3 -MT -Ow -W3 -Zp -Tp
SET LINKER=link
SET GUIFLAGS=-SUBSYSTEM:windows
SET DLLFLAGS=-SUBSYSTEM:windows -DLL
SET GUILIBS=-DEFAULTLIB:user32.lib gdi32.lib winmm.lib comdlg32.lib comctl32.lib
SET RC=rc
SET RCVARS=-r -DWIN32
```

图 1-1 运行这个文件以编译本书中的程序

运行本书附带的软盘中的 Setup.bat ，它将把本书中所用的程序代码拷贝至 C 盘的 petzold 目录下。 MSC.BAT 文件可以在 C：\petzold\Chap01 目录下找到。

我将在后面的章节中说明这些语句的细节。

第 2 章　Hello, Windows 95

如果你对于在 Microsoft Windows 95 这样一个图形环境中编程来说还是一个新手，那么你会发现这种编程不同于你的其它任何经验。Windows 有一个方便用户但不方便程序员的名声。对于初学者来说，不可避免地会被 Windows 的结构和在系统中运行的应用程序的结构所吓倒。如果你也遇到了这种情况，请不要担心，可能是在你的大脑中缺少了什么重要的部分而使你无法成为一个成功的 Windows 程序员。这种最初的迷惑是正常的，不要认为别的人会有什么不同。

基于 Windows 的编程是比较奇怪的。它不太直观，你需要花一些时间才能掌握它。一般估计需要六个月来学习 Windows 的编程。并且在此之后，还需要继续学习。我的唯一希望是本书可以缩短几周或一两个月的学习时间。

你可能会问："既然 Windows 如此困难，为什么还要学呢？"

简单的回答是："你可能没有更多的选择"。总之，PC 市场已经发展到了这样一种程度，使得基于 MS-DOS 的编程，不管是字符界面还是图形界面，都不再适应市场的需要。假设你写了一个商品化的商业软件，如果它不是基于 Windows 的，计算机杂志的软件评论者肯定会对你的产品不予理睬。即使你的编程很有特色，如果你的系统不能和他们使用的已经存在的 Windows 的应用集成在一起，你的用户还有你的雇员都会不高兴的。

当然还有更多的理由可以使我们选择 Windows 作为平台。

2. 1　WINDOWS 的特色

与传统的 MS-DOS 环境相比，Windows 为用户和程序员提供了大量的便利。用户和程序开发人员获得的好处极其相似，因为程序开发人员的工作就是为用户提供他们所需要、所向往的东西。Windows 使人们美梦成真!

2. 1. 1　图形用户界面(GUI)

Windows 是一个图形用户界面(GUI)，有时也称之为"可视化界面"或"图形窗口环境"。这种类型的用户界面概念始于 20 世纪 70 年代中期，先驱性的工作是在 Xerox Palo Alto Research Center(PARC)完成的，当时的工作是围绕 Alto 和 Star 等机器以及 Smalltalk 等环境进行的。这些工作由 Apple 计算机公司引入主流，并使之流行起来。首先用于运气不佳的 Lisa 中；一年以后，又引入 Macintosh，这种 1984 年 1 月推出的新机型无疑要成功得多。

自从 Macintosh 问世以来，图形用户界面就如同雨后春笋，在个人计算机工业界和并

非个人的计算机工业界蓬勃发展起来。按照 Microsoft 的 Charles Simonyi 的说法，图形用户界面显然是目前个人计算机工业界唯一最重要的“交响乐”。

2.1.2 GUI 的概念和基本原理

所有图形用户界面都在位图型的视频显示器上显示图形。图形提供了对屏幕实际能力的较好利用，它是一种表达信息的丰富视觉环境，并且为视频图形和为文档打印而准备的格式文本提供了 WYSIWYG(所见即所得)的可能性。

早期的视频显示器仅用于回显用户由键盘输入的文本。在图形用户界面中，视频显示器成为用户输入的来源。视频显示器以图标和输入设备（按钮和滚动条)的形式显示各种图形对象。用户可以用键盘（或者更直接地用鼠标等定位设备）直接在屏幕上操纵这些对象。可以拖动图形对象、按下鼠标按钮以及滚动滚动条。

因此，用户与程序的交流变得更为密切，这不再是一种信息从键盘到程序，再到视频显示器的单向流动，用户已经能够与显示器上的对象直接交流。

2.1.3 一致的用户界面

用户再也不希望花费很长的时间来学习如何使用计算机以及掌握新程序了。 Windows 对初学者帮助很大，因为所有 Windows 程序都具有基本相同的外观。程序占据一个窗口——屏幕上的一块矩形区域，它由一个标题条标识。程序的大多数功能都是通过程序的菜单引入的。无法在一屏中显示的太多信息可以使用滚动条来看。有些菜单可以激活对话框，用户可以在对话框中输入附加信息。在几乎所有的 Windows 程序中，都有一个用于打开文件的对话框，这个对话框在许多不同的 Windows 程序中看起来都是相同(至少非常相似)的，它几乎总是通过同一菜单选项激活。

一旦学会使用一种 Windows 程序后，那么再学习另一种 Windows 程序就非常容易了。菜单和对话框使用户可以尝试着使用新程序，并发掘其特性。大多数 Windows 程序的大多数功能都可以通过键盘来控制，但使用鼠标要容易得多。

从程序员的观点来看，一致的用户界面是由于使用 Windows 的内部例程构造菜单和对话框而形成的。所有菜单均具有同样的键盘和鼠标界面，因为这项工作是由 Windows 处理的，而不是由应用程序处理的。

2.1.4 多任务的优点

尽管有些人不断询问在单用户计算机上多任务是否必要，但用户确实乐于使用多任务，并且确实从中受益不少。 MS-DOS 的 RAM 驻留程序(如 SIDEKICK)的流行证明了这一点。尽管严格地讲，弹出式程序并不是多任务程序，但它们允许快速的环境切换。这涉及到许多与多任务相同的概念。

在 Windows 下，每个程序实际上都是一个驻留的弹出式程序。用户可以同时显示并运

行几个 Windows 程序。每个程序在屏幕上占据一个矩形窗口。用户可以在屏幕上移动，不同程序之间切换，以及在程序之间传输数据。由于这种显示看起来很像桌面(当然，是在计算机还未主宰办公桌之前的桌面），所以有人用“桌面”来比喻显示多个程序的 Windows 。

Windows 的早期版本使用一个称为“非抢占式”的多任务系统。这意味着 Windows 不利用系统计时器来分配处理时间给系统中运行的多个程序。程序必须自愿放弃控制权，这样其它程序才能运行。而在 Windows 95 中，多任务是抢占式的，而且程序自己也可以分裂为似乎是同时运行着的多个线程。

2.1.5 内存管理

如果不在内存管理方面做一些努力，操作系统是无法实现多任务的。在新程序启动，旧程序终止时，内存会变得零乱。系统必须能将空闲的内存空间组织在一起。这就要求系统移动内存中的数据和代码块。

在 8088 微处理器上运行的 Windows 1 就已经能够实现这种类型的内存管理了。在实模式下，这只能被看成是软件工程的一桩令人惊喜的成就罢了。在 Windows 下运行的程序可以超额调配内存；在任意时间，程序可以包含内存中装不下的代码量。程序可以放弃内存中的代码，然后从程序的.EXE 文件中重新装入。用户可以运行一个程序的多份副本（又叫“实例”）；所有实例共享内存中同样的代码。在 Windows 中运行的程序可以共享位于其.EXE 文件中的例程，这叫做“动态链接库”。 Windows 包含一种技术，能够在运行时将程序与动态链接库中的例程相连接。 Windows 本身就是一个动态链接库的集合。

因此，即使在 Windows 1 中， PC 体系结构的 640KB 内存限制实际上已经被突破了，而且不要求任何附加内存。但 Microsoft 并未就此止步： Windows 2 使得 Windows 应用程序能够访问扩展内存(EMS)； Windows 3 在保护模式下运行，使 Windows 应用程序能够访问高达 16MB 的扩充内存。而如今， Windows 95 完全突破了以前的局限，成为一个提供平面存储空间的 32 位操作系统。

2.1.6 与设备无关的图形界面

Windows 是一个图形界面， Windows 程序能够完全利用视频显示器和打印机上的图形和格式化文本。图形界面不仅在外观上吸引人，而且还能够给用户传递比较高层次的信息。

为 Windows 编写的程序不直接访问屏幕和打印机等图形显示设备硬件。相反，Windows 提供了一种图形程序设计语言（称为图形设备接口，即 GDI)，它使显示图形和格式化的文本更容易。 Windows 虚拟了显示硬件，为 Windows 编写的程序可以在任意视频卡和任意打印机上运行，只要 Windows 有它的设备驱动程序。程序不需要确定系统连接的是哪种设备。

对于 Windows 的开发人员来讲，将一个与设备无关的图形界面输出到 IBM PC 上不是一件简单的事情。 PC 的设计是基于开放体系结构的原则，第三方的硬件制造商被鼓励开发 PC 机的外设，并研制出大量外设。尽管已经出现了一些标准， PC 上的传统 MS-DOS

程序仍然必须分别支持许多不同的硬件配置。例如，对于一个作为商品发售的 MS-DOS 字处理器程序而言，带有一两个软盘的小文件，其中每个文件支持一种特定的打印机，这是非常常见的。 Windows 95 程序不要求这些设备驱动，因为对它们的支持已经成为 Windows 中的组成部分。

2.1.7　投入 Windows

Windows 95 的程序员处于一种要么拥有一切要么一无所有的境地。例如，你无法编写 MS-DOS 应用程序而仅仅为某些图形使用 Windows 。如果你想使用 Windows 的任何部分，那么你就必须保证编写一个成熟的 Windows 程序。

其中的原因将随着你对一个 Windows 程序结构的了解而趋于明朗。在 Windows 中的一切都是相互联系的。如果你想在视频显示器上绘制一些图形，那么你需要一种被称为“设备环境句柄”的东西。为了获取它，你需要一个“窗口句柄”。为了获取窗口句柄，你必须创建一个窗口并准备好接收发送给这个窗口的“消息”。为了接收和处理消息，你需要一个“窗口过程”。此时就需要你编写一个 Windows 程序。你无法飞起来，除非你离开地面。

2.1.8　函数调用

Windows 95 支持一千种以上的函数调用，它们可以在应用程序中使用。要记住这些调用的语法几乎是不可能的。大多数 Windows 程序员花费大量的时间在打印的或者联机的帮助中查找各种的函数调用。

每个 Windows 函数的名字都是描述性的，大小写字母混合书写，如 CreateWindow 。读者可以猜到，这个函数是用来为程序创建一个窗口的。请看另一个例子：函数 IsClipBoardFormatAvailable 是确定剪辑板是否装有某种特定格式的数据。

所有主要的 Windows 函数均在头文件 WINDOWS.H 中有声明，包含在 Windows 软件开发工具包(SDK)和 BORLAND C++编译器中。 WINODWS.H 是 Windows 文档的一个重要部分，可以打印，或者用文件浏览程序进行快速查找。

在 Windows 程序中使用 Windows 函数调用的方法基本上与使用 C 库函数(如 STRLEN)相同。主要的区别在于 C 库函数的代码被连接进你的程序代码中，而 Windows 函数的代码却位于你的程序之外的动态链接库中(DLL)。

当你运行一个 Windows 程序时，它通过一个叫做“动态链接”的过程和 Windows 进行交互。一个 Windows 的.EXE 文件中包含它所使用的函数以及各种动态链接库的引用。其中大多数的 DLL 位于你的 Windows 目录的 SYSTEM 子目录下。在 Windows 程序被装入内存后，程序中的远程调用被转换为指向动态链接库中函数的入口，动态链接库也被装入内存中(如果还没有的话)。

在链接 Windows 程序以产生可执行文件时，必须使用 Windows 软件开发工具包（SDK）或者 BORLAND C++编译程序中提供的专门“载入库（import library）”。这些载入库中包含有动态链接库名和所有 Windows 函数的序数名。链接程序使用这个信息，用来在.EXE 文件中构造一个表，在载入程序时 Windows 使用这个表将调用转换为 Windows

函数。

2.1.9 面向对象的程序设计

进行 Windows 程序设计，实际上就是进行一种面向对象的程序设计(OOP)。这一点在 Windows 中使用得最多的对象身上表现得最为明显。这种对象正是 Windows 之所以命名为“ Windows ”的原因，它具有人格化的特征，甚至可能会在你的梦中出现，这就是那个叫做“窗口”的东西。

我在前面已经提到过，窗口是屏幕上的矩形区域。一个窗口可以从键盘或者鼠标接受用户的输入，并在其内部显示图形输出。

一个应用程序窗口通常包含程序的标题条、菜单、边框，也许还有一些滚动条。对话框也是一种窗口。不同的是，对话框表面通常包含几个其它窗口，称之为“子窗口”。这些子窗口的形式有压入按钮、单选按钮、复选框、文本输入区域、列表框和滚动条。

用户将这些窗口看成屏幕上的对象，可以通过按下一个按钮或者滚动一个滚动条与这些对象直接交互。有趣的是，程序员的观点与用户的观点极其类似。窗口以“消息”的形式接收窗口的输入，窗口也用消息与其它窗口通讯。

对消息的理解将是读者成为 Windows 程序员所必须逾越的障碍之一。

2.1.10 消息驱动式体系结构

作者第一次真实地看到图形用户界面时，很有些困惑不解。我们可以演示一个在窗口中运行的最基本的字处理程序。在程序窗口的大小改变时，字处理器会重新格式化其中的文本。

很明显，窗口大小改变的细节是由操作系统处理的，程序能够响应这个系统功能。程序是怎样“知道”窗口的大小发生改变的呢？操作系统使用了什么技术将这一信息传递给窗口的呢？在苦思冥想这些问题时，我们的传统编程经验似乎毫无用处。

让我们来揭开这个谜底吧！问题的关键在于理解图形用户界面所使用的体系结构。在 Windows 中，当用户改变窗口的大小时，Windows 给程序发送一条消息指出新窗口的大小。然后，程序就可以调整窗口中的内容，以响应大小的变化。

“ Windows 给程序发送消息。”我们希望读者不要对这句话视而不见。它到底表达了什么意思呢？我们在这里讨论的是程序代码，而不是一个电子邮件系统。操作系统怎么能给程序发送消息呢？

其实，所谓“ Windows 给程序发送消息”，是指 Windows 调用程序中的一个函数，该函数的参数描述了这个特定消息。这种位于 Windows 程序中的函数称为“窗口过程”。

2.1.11 窗口过程

无疑，读者对程序调用操作系统的思路是很熟悉的。例如，程序在打开磁盘文件时就

要使用有关的系统调用。读者所不习惯的，可能是操作系统调用程序。而这正是 Windows 面向对象体系结构的基础。

程序创建的每一个窗口都有相关的窗口过程。这个窗口过程是一个函数，既可以在程序中，也可以在动态链接库中。 Windows 通过调用窗口过程来给窗口发送消息。窗口过程根据此消息进行处理，然后将控制返回给 Windows 。

更确切地说，窗口通常是在“窗口类”的基础上创建的。窗口类标识了处理窗口消息的窗口过程。使用窗口类使多个窗口能够基于同一个窗口类，并且使用同一个窗口过程。例如，所有 Windows 程序中的所有按钮均基于同一窗口类。这个窗口类与一个处理所有按钮消息的窗口过程（位于 Windows 的 USER.EXE 动态链接库中）联结。

在面向对象的程序设计中，对象是代码与数据的组合。窗口是一种对象，其代码是窗口过程，数据是窗口过程保存的信息和 Windows 为每个窗口以及系统中那个窗口类保存的信息。

窗口过程处理发送给窗口的消息。这些消息经常是告知窗口，用户正使用键盘或者鼠标进行输入。这正是压入按钮窗口知道它被“按下”的奥妙所在。在窗口大小改变，或者窗口表面需要重画时，由其它消息通知窗口。

Windows 程序开始执行后， Windows 为该程序创建一个“消息队列”。这个消息队列用来存放该程序可能创建的各种不同窗口的消息。程序中有一小段代码，叫做“消息循环”，它用来从队列中取出消息，并且将它们发送给相应的窗口过程。有些消息直接发送给窗口过程，不用放入消息队列中。

如果你对这段有关 Windows 体系结构的过于简略的描述将信将疑，就让我们去看看在实际的程序中，窗口、窗口类、窗口过程、消息队列、消息循环和窗口消息是如何相互配合的。这或许会对你有些帮助。

2.2 编写第一个 Windows 程序

在经典著作《C 程序设计语言》(Prentice Hall 1988 年第二版)中， Brian Kernighan 和 Dennis Ritchie 开始使用一个现在已经非常有名的“ HELLO, WORLD ”程序来讨论 C：

```
#include <stdio.h>

main ( )
     {
     printf("Hello, world\n") ;
     }
```

在本章中，我们将给出为 Microsoft Windows 编写的类似的程序。我们称这个程序为 HELLOWIN 。它创建一个窗口，并在窗口中显示“ Hello ， Windows!”，同时播放一个声音文件，用我的声音朗读同样的一句话。

为了避免读者第一次看到 HELLOWIN 时由于过于震惊而打退堂鼓，我们必须先警告大家， HELLOWIN.C 的源代码长度在 80 行以上，而且，这 80 多行中大多数都是些相对

固定的准备工作。在几乎所有 Windows 程序中，都会有类似的准备工作。

与其询问 " Hello, Windows 95!" 程序为何如此冗长复杂，不如先来看一看传统的 " Hello, World " 程序为何如此短小和简单。

2.2.1 传统程序的问题何在

" Hello, World " 程序以及其它传统 C 程序的输出模型是过时的硬件，即电传打字机。电传打字机用连续的卷筒纸模拟打字机。在不久以前，程序员还坐在电传打字机旁输入命令，命令回显在纸上。计算机通过将输出打印在纸上来响应输入命令。

在 Mainframe（大型机）终端和个人计算机的早期，电传打字机模型就已经扩展为视频显示器。视频显示器可以说是一种"玻璃电传打字机"。当文本到达屏幕底部时，它只是滚动一下。

为什么传统的 " Hello, World " 程序不用告诉操作系统文本在什么样的特定输出设备上显示，但却能显示文本呢？因为只有一台输出设备，视频显示器就被当成电传打字机来使用。如果用户希望输出到其它设备上，则必须在命令行中重新定向。

程序怎样才能在不告诉系统文本出现在输出设备的什么地方的情况下来显示文本呢？由于文本总是在光标当前所在的地方开始显示，多半是从执行程序的下一行开始。假如用户想在屏幕中间显示 " Hello, World "，则必须使用与设备无关的控制码，先将光标定位在期望的地方。

假设用户想运行几次 " Hello, World " 程序，看一看它们在屏幕上的输出。简直太乱了！程序之间会相互影响，在电传打字机模型中，没有什么机制能够将并发运行的程序的输出分开。

程序结束之后，" Hello, World " 输出还停留在屏幕上，这也很有趣。它结束之后没有清除输出。而是在视频显示器上留下了象征它存在的遗迹。

" Hello, World " 太简单了，因为它是为一个简单的时代，简单的计算机和简单的输出设备而设计的。它与我们今天所思考的现代软件不在同一水平线上，几乎具有天壤之别，不可同日而语。

2.2.2 HELLOWIN 的文件

创建 " Hello， Windows " 程序必须有三个文件，其中的两个如图 2-1 所示，一个是 HELLOWIN.MAK "制作" 文件，另一个是 HELLOWIN.C 源代码文件。第三个文件是 HELLOWIN.WAV，可以在用本书所附软盘安装后的 Chap02 子目录下找到。这是一个包含朗读正文的声波文件。

HELLOWIN.MAK

```
#------------------------
# HELLOWIN.MAK make file
#------------------------
```

```
hellowin.exe : hellowin.obj
     $(LINKER) $(GUIFLAGS) -OUT:hellowin.exe hellowin.obj $(GUILIBS)

hellowin.obj : hellowin.c
     $(CC) $(CFLAGS) hellowin.c
```

HELLOWIN.C

```
/*------------------------------------------------------------
   HELLOWIN.C -- Displays "Hello, Windows 95!" in client area
                 (c) Charles Petzold, 1996
  ------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[ ] = "HelloWin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName,         // window class name
                    "The Hello Program",     // window caption
                    WS_OVERLAPPEDWINDOW,     // window style
                    CW_USEDEFAULT,           // initial x position
                    CW_USEDEFAULT,           // initial y position
                    CW_USEDEFAULT,           // initial x size
                    CW_USEDEFAULT,           // initial y size
                    NULL,                    // parent window handle
                    NULL,                    // window menu handle
                    hInstance,               // program instance handle
                    NULL) ;                  // creation parameters

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
```

```
     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_CREATE :
               PlaySound ("hellowin.wav", NULL, SND_FILENAME | SND_ASYNC) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               DrawText (hdc, "Hello, Windows 95!", -1, &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }

     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 2-1 HELLOWIN 程序

在第 9 章中，你将遇到另一类在 Windows 编程中常见的文件类型，称为“资源文件”，以.RC 作为文件扩展名。但目前为止，大多数示例程序将简单地使用一个制作文件和一个 C 源代码文件，可能再有一个头文件。

我在前面已经提到过， HELLOWIN.C 中大多数行都是些相对固定的准备工作，实际上几乎每个 Windows 程序中都有这种情况。没有人能真正记住书写这些语句的所有语法。一般来说， Windows 程序员在开始书写一个新程序之前，都会先拷贝一个已经存在的程序并恰当地进行一些修改。你可以自由地使用本书所附的软盘中的程序。

如果你安装了 Windows 95 和 Microsoft Visual C++ 4.0 ，并且有 Visual C++中所带的 VCVARS32.BAT 批处理文件和第 1 章中所示的 MSC.BAT 批处理文件，那么你应该可以通过在 MS-DOS 命令行上运行:MAKE HELLOWIN.MAK 来创建 HELLOWIN.EXE 。

如果一切顺利的话，你可以简单地从 MS-DOS 命令行输入:

```
HELLOWIN
```

从而运行该程序。

程序创建一个普通的应用程序窗口，如图 2-2 中所示。在窗口客户区域的中央显示“ Hello ， Windows 95!”。如果你安装了一个声卡，那么你还可以听到声音信息(如果你没有声卡，那你还等待什么呢？)。

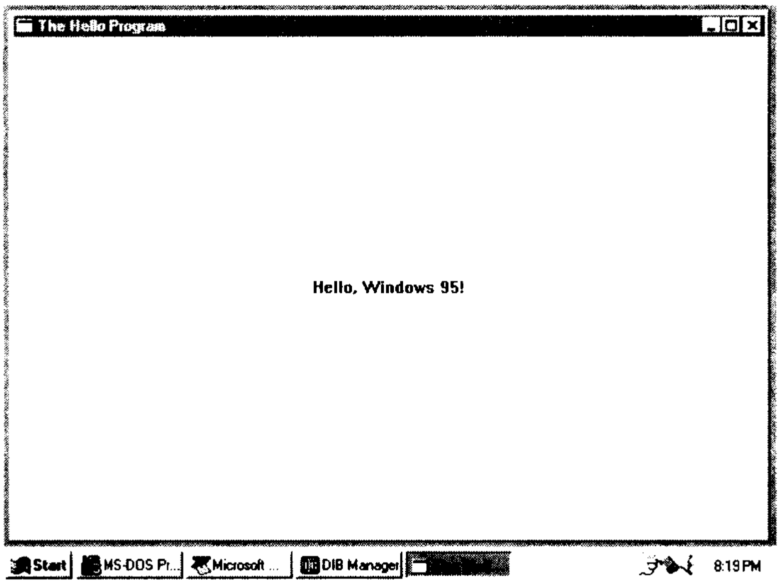


图 2-2　在Windows 95下运行的HELLOWIN

如果你认真思考一下，将会发现，虽然只有 80 行代码，这个窗口却令人惊讶地具有许多的功能。你可以用鼠标按住标题条，在屏幕上移动窗口；可以按住大小边框，改变窗口的大小。在窗口大小改变时，程序自动地将“ Hello ， Windows 95!”文本串重新定位在客户区域的中央。你可以按最大化按钮，放大 HELLOWIN 以充满整个屏幕；也可以按最小化按钮，将程序压缩为一个图标。你可以在系统菜单中执行所有选项；也可以通过从系统菜单中选择终止选项，或者通过点一下标题条最右端的关闭按扭，或者通过双点标题条最左端的图标来关闭窗口以终止程序的执行。

读者看到 HELLOWIN 具有普通 Windows 程序的所有功能时，当然会很高兴；但当读者研究创建程序所需的源代码时，就不一定会那么高兴了。虽然如此，我们还是要勇敢地逐条语句地彻底剖析这个程序。

2. 2. 3　Make 文件

为了简化 Windows 程序的编译，可以使用 Microsoft Visual C++ 4.0 版本中提供的 NMAKE 工具。每当改变了 HELLOWIN.C 源代码文件的内容后，所需做的就是运行如上所示的 NMAKE ，以创建最新版本的 HELLOWIN.EXE 可执行文件。

一个 MAKE 文件包含一个或多个段，每段以一个左顶格行开始，列出一个目标文件，

后跟一个冒号，再跟一个或多个目标文件的依赖文件。这一行后跟一行或多行缩进命令。这些命令根据依赖文件来创建目标文件。如果任何一个依赖文件的最后一次修改日期和时间比目标文件的最后一次修改日期和时间晚，则 NMAKE 就会执行缩进的命令行。

通常，NMAKE 只更新 make 文件第一段中的目标文件。不过，如果依赖文件本身是 make 文件另一段中的目标文件，那么 NMAKE 将先更新这个目标文件。

HELLOWIN.MAK 制作文件包含两段。如果 HELLOWIN.OBJ 或者 HELLOWIN.DEF 在 HELLOWIN.EXE 之后修改过，那么第一段将运行链接程序。如果在生成 HELLOWIN.OBJ 之后改变过 HELLOWIN.C ，则第二段运行 C 编译程序。由于 HELLOWIN.OBJ 既是制作文件第一段中的依赖文件，又是第二段中的目标文件，所以 NMAKE 将会检查在重新创建 HELLOWIN.EXE 之前是否需要重新更新 HELLOWIN.OBJ 。因此，制作文件的分析是从底部向上进行的。

在第 1 章中，讨论过如何通过批处理文件设置环境变量，从而为制作文件提供宏标识符。这通常包括设置各种编译标志和链接库名。如果你对这些细节有兴趣，可以回到第 1 章中的相关部分。

2.2.4 C 源代码文件

图 2-1 中的第三个文件 HELLOWIN.C 是源代码文件。可能有些读者一开始还不敢相信这个程序真是用 C 编写的。

在详细讨论之前，我们先大致看一看这个程序。这个文件只包含两个函数:WinMain 和 WndProc 。WinMain 是程序的入口点，等价于标准 C 的 main 函数。每个 Windows 程序都有一个 WinMain 函数。

WndProc 是 HELLOWIN 窗口的窗口过程，该函数处理发送给窗口的消息。每个窗口——无论是像应用程序的主窗口那样的大窗口还是像按钮那样的小窗口——都与窗口过程相关。这些窗口过程是一种代码封装方式，它们对输入（一般是由键盘或鼠标产生）进行响应，然后把结果以图形方式显示到屏幕上。正如我们将要看到的，窗口过程是通过对发送给窗口的“消息”进行处理来完成以上功能的。不过目前您还不需要担心它的工作原理，因为后面您会有大量的时间来熟悉这些概念。

在 HELLOWIN.C 中没有直接调用 WndProc 代码， WndProc 只供 Windows 调用。但是，在 WinMain 中有一个 WndProc 引用，这就是在程序开头，在 WinMain 之前声明该函数的原因。

2.2.5 Windows 函数调用

HELLOWIN 至少调用了 16 个 Windows 函数。下面以它们在 HELLOWIN 中出现的次序列出这些函数以及各自的简明描述:

- LoadIcon——装入图标供程序使用
- LoadCursor——装入鼠标光标供程序使用
- GetStockObject——获取一个图形对象（在这个例子中，是获取绘制窗口背景的刷子）

- RegisterClass——为程序窗口注册窗口类
- CreateWindow——根据窗口类创建一个窗口
- ShowWindow——在屏幕上显示窗口
- UpdateWindow——指示窗口刷新自身
- GetMessage——从消息队列中获取消息
- TranslateMessage——转换某些键盘消息
- DispatchMessage——将消息发送给窗口过程
- PlaySound——播放一个声音文件
- BeginPaint——初始窗口绘制
- GetClientRect——获取窗口客户区域的尺寸
- DrawText——显示文本串
- EndPaint——结束窗口绘制
- PostQuitMessage——在消息队列中插入一条“退出”消息
- DefWindowProc——执行缺省的消息处理

这些函数均载入《Windows 程序员参考手册》，并在 WINDOWS.H 中声明。

2.2.6 大写字母标识符

读者可能注意到，HELLOWIN.C 中有几个大写的标识符，这些标识符是在 WINDOWS.H 中定义的。有些标识符含有两个字母或者三个字母的前缀，后跟一个下划线：

CS_HERDRAW	DT_VCENTER	WM_CREATE
CS_VREDRAW	IDC_ARROW	WM_DESTORY
CW_USEDEFAULT	IDI_APPLICATION	WM_PAINT
DT_CENTER	SND_ASYNC	
WS_OVERLAPPEDWINDOW		
DT_SINGLELINE	SND_FILENAME	

这些是简单的数字常数。前缀指示该常量所属的类别，如下表所示：

前缀	类别
CS	类风格
IDI	图标 ID
IDC	光标 ID
WS	窗口风格
CW	创建窗口
WM	窗口消息
SND	声音
DT	绘制文本

奉劝程序员不要费力气去记忆 Windows 编程中的数值常量。实际上， Windows 中使用的每个数值常量在 WINDOWS.H 中均有相应的标识符定义。

2.2.7 新的数据类型

HELLOWIN.C 中的其它标识符是新的数据类型，也在 WINDOWS.H 中使用 typedef 语句或者#defind 语句加以了定义。这最初是为了便于将 Windows 程序从原来的 16 位系统上移植到未来的基于 32 位(或者其它)技术的操作系统上。这种作法并不如当时每个人想象的那么顺利，但是这种概念基本上是正确的。

有时这些新的数据类型只是为了方便缩写。例如，用于 WndProc 的第二个参数的 UINT 数据类型只是一个 unsigned int （无符号整数），在 Windows 95 中，这是一个 32 位的值。用于 WinMain 的第三个参数的 PSTR 数据类型是指向一个字符串的指针，即是一个 char *。

其它数据类型的含义不太明显。例如， WndProc 的第三和第四个参数分别被定义为 WPARAM 和 LPARAM 。这些名字的来源有点历史背景：当 Windows 是一种 16 位系统时，WndProc 的第三个参数被定义为一个 WORD ，这是一个 16 位的无符号短整数，而第四个参数被定义为一个 LONG ，这是一个 32 位的有符号长整数，从而导致了在单词“ PARAM ”的前缀“ W ”和“ L ”。在 Windows 95 中， WPARAM 被定义为一个 UINT ，而 LPARAM 被定义为一个 LONG （这就是 C 中的 long 整数类型），因此窗口过程的这两个参数都是 32 位的值。这也许有点奇怪，因为 WORD 数据类型在 Windows 95 中仍然被定义为一种 16 位的无符号整数，因此“ PARAM ”的“ W ”在某种意义上是不正确的名字。

WndProc 函数返回一个类型为 LRESULT 的值，该值简单地被定义为一个 LONG 。WinMain 函数被指定了一个 WINAPI（在头文件中定义的所有 Windwos 函数都被指定这种类型）类型，而 WndProc 函数被指定一个 CALLBACK 类型。这两个标识符都被定义为__stdcall ，表示在 Windows 本身和用户的应用程序之间发生的函数调用的调用序列。

HELLOWIN 还使用了 Windows 头文件中定义的四种数据结构(我们将在本章稍后加以讨论)。这些数据类型是：

结构	含义
MSG	消息结构
WNDCLASSEX	窗口类结构
PAINTSTRUCT	绘图结构
RECT	矩形结构

前两个数据结构在 WinMain 中使用，分别定义了两个名为 msg 和 wndclass 的结构，后面两个数据结构在 WndProc 中使用，分别定义了 ps 和 rect 结构。

2.2.8 句柄简介

最后，还有三个大写标识符，用于不同类型的“句柄”：

标识符	含义
HINSTANCE	实例(程序自身)句柄
HWND	窗口句柄
HDC	设备环境句柄

句柄在 Windows 中使用非常频繁。在本章结束之前，我们将遇到 HICON(图标句柄)、HCURSOR(鼠标光标句柄)和 HBRUSH(图形刷子句柄)。

句柄是一个（通常为 32 位的）数，它代表一个对象。 Windows 中的句柄类似传统 C 或者 MS-DOS 程序设计中使用的文件句柄。程序几乎总是通过调用 Windows 函数获取句柄。程序在其它 Windows 函数中使用这个句柄，以引用它代表的对象。句柄的实际值对程序来说是无关紧要的。但是，向你的程序提供句柄的 Windows 模块知道如何使用它来引用相应的对象。

2.2.9 匈牙利表示法

读者可能注意到， HELLOWIN.C 中有一些变量的名字显得很古怪。如 szCmdLine ，它作为参数传递给 WinMain 。

许多 Windows 程序员使用一种叫做“匈牙利表示法”的变量命名约定。这是为了纪念传奇性的 Microsoft 程序员 Charles Simonyi 。非常简单，变量名以一个或者多个小写字母开始，这些字母表示变量的数据类型。例如， szCmdLine 中的 sz 代表“以 0 结尾的串”。在 hInstance 和 hPrevInstance 中的 h 前缀表示“句柄”；在 iCmdShow 中的 i 前缀表示“整数”。 WndProc 的后两个参数也使用匈牙利表示法。正如我在前面已经解释过的，尽管 wParam 应该更适当地被命名为 uiParam(代表“无符号整数”)，但是因为这两个参数是使用数据类型 WPARAM 和 LPARAM 定义的，因此保留它们传统的名字。

在命名结构变量时，可以用结构名(或者结构名的一种缩写)的小写作为变量名的前缀，或者用作整个变量名。例如，在 HELLOWIN.C 的 WinMain 函数中， msg 变量是 MSG 类型的结构； wndclass 是 WNDCLASSEX 类型的一个结构。在 WndProc 函数中， ps 是一个 PAINTSTRUCT 结构， rect 是一个 RECT 结构。

匈牙利表示法能够帮助程序员及早发现并避免代码中的错误。由于变量名既描述了变量的作用，又描述了其数据类型，就比较容易避免产生数据类型不匹配的错误。

下表列出了在本书中将经常使用的变量前缀：

前缀	数据类型
c	char
by	BYTE(无符号字符)
n	short 或者 int
i	int(整数)
x， y	short(分别用作 x 坐标和 y 坐标)

续表

前缀	数据类型
cx， cy	short(分别用作 x 长度和 y 长度; C 代表 “ count ”)
b 或 f	BOOL(int)
w	WORD(无符号 short)
l	LONG(长整数)
dw	DWORD(无符号长整数)
fn	function(函数)
s	string(串)
sz	以 0 字节结尾的串
h	句柄
p	指针

2.2.10 程序的入口点

经过对 HELLOWIN.C 的大致分析之后，我们可以开始一行一行的剖析这个程序了。代码以#include 语句开始，这条语句用来包含 WINDOWS.H 头文件。

```
#include <windows.h>
```

WINDOWS.H 包含了对 Windows 函数、 Windows 结构、新的数据类型和数值常量的声明。

其后跟一个对 WndProc 函数的预先声明：

```
LRESULT CALLBACK WndProc(HWND, UNIT, UNIT, LONG) ;
```

之所以需要预先声明，是因为在 WinMain 函数中有代码引用 WndProc 。

在为传统环境编写的 C 程序中，入口点是 main 函数。这就是程序开始执行的地方(实际上， main 函数是程序员编写的程序的入口点。通常， C 编译程序将在可执行文件中插入一些启动代码，由启动代码来调用 main)。 Windows 程序的入口点是 WinMain 函数(与 main 的情况相同， WinMain 实际也是由插入到可执行文件中的启动代码调用的)。WinMain 通常定义为如下形式:

```
int WINAPI WinMain(HANDLE hInstance , HANDLE hPrevInstance ,
                   PSTR lpzCmdParam int nCmdShow) ;
```

这个函数使用 PASCAL 调用序列，并返回一个整数给启动代码。函数必须命名为 WinMain ，它有四个参数。

hInstance 参数是所谓的“实例句柄” 。它是一个数值；当程序在 Windows 下运行时，它用于唯一地标识该程序。可能会存在这种情况：在 Windows 下运行同一程序的几个副本。例如， Windows 的很多用户不时地加载多个 CLOCK 程序。我们看一看这会发生什么情况。每次运行一个“实例”，每个实例均有不同的 hInstance 值。可以将实例句柄与多任务操作系统中很常见的“任务 ID ”或者“进程 ID ”进行类比。

hPrevInstance("前一个实例")参数是同一程序的现在仍处于活动状态的最近一个实例的句柄。如果当时未装入该程序的其它副本，则hPrevInstance将为0或为NULL。在Windows 95 中，该参数总是为 NULL。

szCmdLine 参数是一个指针，指向一个以 0 终结的串，串中包含有传给该程序的命令行参数。用户可以在程序管理器或者文件管理器里将程序名和参数输入到 Run 对话框中，以此来运行带命令行参数的 Windows 程序。

iCmdShow 参数是一个数值，指示窗口将如何在 Windows 中被显示。这个数值由在 Windows 下运行该程序的程序所指定。程序并不经常需要查看这个数值，但是在需要的时候可能会查看它。在大多数情况下，这个数值不是 1 就是 7；但最好不要简单地考虑为 1 或者 7，而是使用常量 SW_SHOWNORMAL(在 WINDOWS.H 中定义为 1)或者 SW_SHOWMINNOACTIVE(定义为 7)。这两个标识的 SW 前缀代表"显示窗口"，它们指示程序开始时是显示普通窗口还是显示最小化窗口。

2.2.11 注册窗口类

窗口总是在窗口类的基础上创建的，窗口类用以标识处理窗口消息的窗口过程。这是很重要的。我们再不厌其烦地重复一遍：窗口总是在窗口类的基础上创建的，窗口类用以标识处理窗口消息的窗口过程。

可以在单个窗口类的基础上创建多个窗口。例如，Windows 中的所有按钮窗口都是基于同一窗口类创建的。窗口类定义了窗口过程和基于此类创建的窗口的其它一些特征。在创建窗口时，要定义一些窗口所独有的特征。

在为程序创建窗口之前，必须首先调用 RegisterClassEx 注册一个窗口类。该函数是 Windows 早期版本中的 RegisterClass 函数的扩充版本，不过在 Windows 95 中，RegisterClass 仍然是有效的。

RegisterClassEx 函数只需要一个参数，即一个指向类型为 WNDCLASSEX 的结构指针。WNDCLASSEX 结构在 WINDOWS.H 中的定义为:

```
typedef struct tagWNDCLASSEX
    {
    UINT        cbSize;
    UINT        style;
    WNDPROC     lpfnWndProc;
    int         cbClsExtra;
    int         cbWndExtra;
    HINSTANCE   hInstance;
    HICON       hIcon;
    HCURSOR     hCursor;
    HBRUSHhbrBackground;
    LPCSTRlpszMenuName;
    LPCSTRlpszClassName;
    HICON       hIconSm;
    }
    WNDCLASSEX;
```

在这里提示一下数据类型和匈牙利表示法：其中的 LP 和 lp 前缀代表“长指针”，这是 16 位 Windows 的遗物，那时程序员必须区别短(或者近)16 位指针和长(或者远)32 位指针。在 Windows 95 中，所有指针的大小都是 32 位，我曾经试图在本书的示例程序中去掉指针类型中的 l 前缀，但是你不可避免地会在其它的地方看到它们。

你还会注意到匈牙利表示法的一些其它用法： lpfn 前缀代表“指向一个函数的长指针”。 cb 前缀代表“字节数”。 hbr 前缀代表“一个刷子的句柄”。

在 WinMain 中，必须定义一个类型为 WNDCLASSEX 的结构，如下所示：

```
WNDCLASSEX wndclass;
```

然后，可以确定该结构的十二个域，并调用 RegisterClassEx:

```
RegisterClassEx (&wndclass) ;
```

最重要的两个域是倒数第二和第三个，倒数第二个域是窗口类名 (在只创建一个窗口的程序中，通常设为与程序名相同)，倒数第三个域 (lpfnWndProc)是基于这个类来创建的所有窗口所使用的窗口过程的地址 (在 HELLOWIN.C 中，这个过程是 WndProc 函数)。所有其它的域描述了基于该窗口类的所有窗口的特征。

cbSize 域是结构的大小。语句:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW;
```

使用 C 的按位“或”操作符组合了两个“类风格”标识符。在 WINDOWS.H 中，以 CS 为前缀开头的标识符定义为 16 位常量，其中只有一位置位。例如， CS_VREDRAW 定义为 0x0001， CS_HREDRAW 定义为 0x0002。用这种方式定义的标识符被称为“位标志”。程序员可以使用位“或”操作符来组合位标志标识符。

这两个类风格标识符表明，对所有给予此类风格的窗口，每当窗口在水平方向或者垂直方向的尺寸改变之后，要完全刷新窗口。改变 HELLOWIN 的窗口大小，可以看到文本串仍然显示在窗口的中央。这两个标识符确保了这一点。不久我们就将看到窗口过程是如何被告知这种窗口大小的变化的。

WNDCLASSEX 结构中的第三个域由下列语句进行初始化:

```
wndclass.lpfnWndProc = WndProc;
```

这条语句将这个窗口类的窗口过程设置为 WndProc，即 HELLOWIN.C 中的第二个函数。这个过程将处理给予这个窗口类创建的所有窗口的所有消息。我已经提到过，域名中的 lpfn 前缀是匈牙利表示法，代表“ long ponter to a function (指向函数的长指针)”。

下面两条语句:

```
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
```

在类结构中和 Windows 内部保存的窗口结构中预留额外空间。程序可以根据需要来使

用预留的空间。 HELLOWIN 没有使用它们，所以赋值为 0 。否则，和匈牙利表示法所指示的一样，这个域将被置为“字节数”。

下一个域就是程序的实例句柄 (它也是 WinMain 的参数之一):

```
wndclass.hInstance = hInstance;
```

语句:

```
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION) ;
```

和语句

```
Wndclass.hIconSm=LoadIcon(NULL,IDI_APPLICATION);
```

为所有基于这个窗口类建立的窗口设置一个图标。图标是一个小的位图图象，将出现在 Windows 任务条中和窗口的标题条的左端。在本书的后面，你将学习如何为你的 Windows 程序定制图标。现在，为了方便起见，我们将使用预定义的图标。

要获取预定义图标的句柄，可以将第一个参数设置为 NULL 来调用 LoadIcon 。在装入程序员定制的图标时，这个参数应该被设置为程序的实例句柄。第二个参数是以 IDI 开始的标识符， IDI 代表“ ID for an icon (图标 ID)”，标识符在 WINDOWS.H 中定义。IDI_APPLICATION 图标只是一个空白方块，其边界为黑色。 LoadIcon 函数返回该图标的句柄。我们并不关心这个句柄的实际值，它只用于设置 hIcon 和 hIconSm 域的值。 hIcon 域在 WNDCLASSEX 结构中定义为 HICON 类型，此类型名的含义为“ handle to an icon ”(图标句柄)”。

语句:

```
wndclass.hCursor = LoadCursor (NULL , IDC_AROW) ;
```

与前一条语句非常相似，LoadCursor 函数装入一个预定义的鼠标光标（命名为 IDC_ARROW)，并返回该光标的句柄。该句柄被赋给 WNDCLASSEX 结构的 hCursor 域。当鼠标光标在基于这个类创建的窗口的客户区域上出现时，它变成一个小箭头。

下一个域指定基于这个类创建的窗口背景颜色。 hbrBackground 域名中的 hbr 前缀代表“ handle to a brush ” (刷子句柄)。刷子是图形学上的一个术语，指用来填充一个区域的着色象素模式。 Windows 有几个标准刷子，也称为“备用（ stock ）”刷子。这里所示的 GetStockObject 调用将返回一个白色刷子的句柄:

```
wndclass.hbrBackground = GetStockObject (WHITE_BRUSH) ;
```

这意味着窗口客户区域的背景完全为白色，这是一种极其普遍的做法。

下一个域指定窗口类菜单。 HELLOWIN 没有应用程序菜单，所以该字段被设置为 NULL:

```
wndclass.lpzMenuName = NULL;
```

最后，必须给出一个类名.类名与程序名相同(即“ Hellowin ”串，它存放在 szAppName 变量中):

```
wndclass.lpszClassName = szAppName;
```

在初始化完结构的 12 个域后， HELLOWIN 调用 RegisterClassEx 来注册这个窗口类。该函数只有一个参数，即指向 WNDCLASSEX 结构的指针：

```
RegisterClassEx (&wndclass) ;
```

2.2.12 创建窗口

窗口类定义了窗口的一般特征，因此可以使用同一窗口类创建许多不同的窗口。实际调用 CreateWindow 创建窗口时，可以指定有关窗口的更详细的信息。

新的 Windows 程序员有时会混淆窗口类和窗口之间的区别，以及为什么一个窗口的所有特征不能被一步到位地指定。实际上，以这种方式分开这些风格信息是非常方便的。例如，所有的按钮窗口都可以基于同样的窗口类来创建。与这个窗口类相关的窗口过程位于 Windows 内部。由窗口类来负责处理按钮的键盘和鼠标输入，并定义按钮在屏幕上的外观形象。从这一点看来，所有的按钮都是以同样的方式工作的。但是并非所有的按钮都是一样的。它们可以有不同的大小，不同的屏幕位置，以及不同的正文串。后面的这样一些特征是窗口定义的一部分。

CreateWindows 与 RegisterClass 不同，它不是使用一个结构，而是要求所有信息都作为参数传递给函数，下面是 HELLOWIN.C 中的 CreateWindow 调用：

```
hwnd = CreateWindow (szAppName, //窗口类名
    "The Hello Program",        //窗口标题
    WS_OVERLAPPEDWINDOW,        //窗口风格
    CW_USEDEFAULT,              //初始 x 位置
    CW_USEDEFAULT,              //初始 y 位置
    CW_USEDEFAULT,              //初始 x 大小
    CW_USEDEFAULT,              //初始 y 大小
    NULL,                       //父窗口句柄
    NULL,                       //窗口菜单句柄
    hInstance,                  //程序实例句柄
    NULL) ;                     //创建参数
```

为了便于你的阅读，我使用了//作为单行注释符号。以上注释分别描述了 CreateWindow 函数的对应参数行。

注释为“窗口类名”的参数为 szAppName ，它包含“ HelloWin ”串，是我们刚刚登录的窗口类名。这是窗口与窗口类相联系的途径。

此程序创建的窗口是一个普通的重叠式窗口，它含有一个标题条，标题条左边有一个系统菜单框，标题条右边有缩小、放大和关闭图标，四周还有一个表示窗口大小的边框。这是标准风格的窗口，在 WINDOWS.H 中定义为 WS_OVERLAPPEDWINDOW ，该参数在程序中注释为“窗口风格”。“窗口标题”是显示在标题条中的文本。

注释为“初始 x 位置”和“初始 y 位置”的参数指定了窗口左上角相对于屏幕左上角的初始位置。在该程序中，这两个参数使用的都是 WS_USEDEFAULT 标识符，指示使用重叠窗口的缺省位置 (CW_USEDEFAULT 在 WINDOWS.H 中定义为 0x80000000)。在缺省情况下， Windows 依次定位重叠式窗口，使各窗口与屏幕左上角的垂直和水平距离分别

成等差数列。与此类似，“初始 x 大小”和“初始 y 大小”参数分别指定窗口的宽度和高度。同样使用了 CW_USEDEFAULT 标识符，表明希望使用缺省尺寸。

注释为“父窗口句柄”的参数设置为 NULL，因为该窗口没有父窗口（如果在窗口之间存在父子关系，则子窗口总是出现在父窗口的上面）。“窗口菜单句柄”也设置为 NULL，因为此窗口没有菜单。“窗口实例句柄”设置为实例句柄，它是作为 WinMain 的参数传递给这个程序的。最后，“创建参数”指针设置为 NULL，可以用这个指针访问以后想要引用的程序中的数据。

CreateWindow 调用返回被创建的窗口的句柄，该句柄存放在变量 hwnd 中，后者被定义为 HWND 类型（窗口句柄类型）。Windows 中的每个窗口都有一个句柄，程序用句柄来引用窗口。许多 Windows 函数需要使用 hwnd 作为参数，这样，Windows 才能知道函数是针对哪个窗口的。如果一个程序创建了许多窗口，则每个窗口均有一个句柄。窗口句柄是 Windows 程序处理的最重要的句柄之一。

2.2.13 显示窗口

在 CreateWindow 调用返回之后，Windows 内部已经创建了这个窗口。但是，窗口并未在视频显示器上出现，还需要两个调用，一个是：

```
ShowWindow (hwnd, iCmdShow) ;
```

第一个参数是刚刚用 CreateWindow 创建的窗口句柄。第二个参数是作为参数传给 WinMain 的 nCmdShow，它确定最初如何在屏幕上显示窗口。当它等于 SW_SHOWNORMAL（等于 1）时，窗口被正常显示。如果 nCmdShow 等于 SW_SHOWMINNOACTIVE（等于 7），则不显示而只是让它的名字和图标出现在任务栏上。

ShowWindow 函数在显示器上显示窗口或者图标。如果 ShowWindow 的第二个参数是 SW_SHOWNORMAL，则窗口的客户区域被窗口类中定义的背景刷子所覆盖。

函数调用：

```
UpdateWindow (hwnd) ;
```

导致客户区域被绘制，它通过给窗口过程（即 HELLOWIN.C 中的 WndProc 函数）发送一条 WM_PAINT 消息做到这一点。后面，我们将说明 WndProc 如何处理这条消息。

2.2.14 消息循环

调用 UpdateWindow 之后，窗口就出现在视频显示器上。程序现在必须准备读入用户用键盘和鼠标输入的数据。Windows 为当前运行的每个 Windows 程序维护一个“消息队列”。在发生输入事件之后，Windows 将事件转换为一个“消息”，并将消息放入程序的消息队列中。

程序通过执行一块称之为“消息循环”的代码从消息队列中取出消息：

```
while (GetMessage (&msg , NULL , 0 , 0))
    {
    TranslateMessage  (&msg) ;
    DispatchMessage   (&msg) ;
    }
return msg.wParam;
```

msg 变量是类型为 MSG 的结构，类型 MSG 在 WINDOWS.H 中定义如下:

```
typedef struct tagMSG
    {
    HWND        hwnd ;
    UINT        message ;
    WPARAM      wParam ;
    LPARAM      lParam ;
    DWORD       time ;
    POINT       pt ;
    }
    MSG ;
```

POINT 数据类型也是一个结构，其定义为:

```
typedef struct tagPOINT
    {
    int x ;
    int y ;
    }
    POINT ;
```

消息循环以 GetMessage 调用开始，它从消息队列中取出一条消息:

```
GetMessage (&msg, NULL, 0, 0) ;
```

这一调用传给 Windows 一个指向名为 msg 的 MSG 结构的指针。第二、第三和第四个参数设置为 NULL 或者 0 ，表示程序接收它自己创建的所有窗口的所有消息。 Windows 用从消息队列中取出的下一条消息来填充消息结构的各个域，结构的各个域包括:

- hwnd——消息发向的窗口的句柄。在 HELLOWIN 程序中，这一参数与 CreateWindow 返回的 hwnd 值相同，因为这是该程序拥有的唯一窗口。
- message——消息标识符。这是一个数值，用以标识消息。对于每条消息，均有一个对应的标识符，这些标识符定义于 WINDOWS.H 中，以前缀 WM (“ window message ”，窗口消息)开头。例如，用户将鼠标光标放在 HELLOWIN 客户区域之内，并按下鼠标左按钮， Windows 就在消息队列中放入一条消息，该消息的 message 域等于 WM_BUTTONDOWN 。这是一个常量，其值为 0x0201 。
- wParam——一个 32 位的“ message parameter (消息参数)”，其含义和数值根据消息的不同而不同。
- lParam——一个 32 位的消息参数，其值与消息有关。
- time——消息放入消息队列中的时间
- pt——消息放入消息队列时的鼠标坐标。

只要从消息队列中取出消息的 message 域不为 WM_QUIT (0x0012)， GetMessage 就返回一个非零值。若消息为 WM_QUIT，则将导致程序退出消息循环，然后程序终止，返回 msg 结构的 wParam 参数。

语句:

```
TranslateMessage (&msg) ;
```

将 msg 结构传给 Windows，进行一些键盘转换。关于这一点，我们将在第三章中深入讨论。

语句:

```
DispatchMessage (&msg) ;
```

又将 msg 结构回传给 Windows。然后， Windows 将该消息发送给适当的窗口过程，让它进行处理。在 HELLOWIN 中，这个窗口过程就是 WndProc 函数。处理完消息之后，WndProc 返回到 Windows。此时， Windows 还停留在 DispatchMessage 调用中。在结束 DispatchMessage 调用的处理之后， Windows 回到 HELLOWIN，并且接着从下一个 GetMessage 调用开始消息循环。

2.2.15 窗口过程

以上我们所讨论的仅仅只是准备性的工作：注册窗口类，创建窗口，然后在屏幕上显示窗口，程序进入消息循环，从消息队列中取出一条消息。

实际的动作发生在窗口过程中。 Windows 程序员通常称窗口过程为“ window proc ”(读作“ prock ”)。窗口过程确定了在窗口的客户区域中显示些什么以及窗口怎样响应用户输入。

在 HELLOWIN 中，窗口过程是命名为 WndProc 的函数。窗口过程可任意命名 (只要不和其它名字发生冲突)。一个 Windows 程序可以包含多个窗口过程。一个窗口过程总是与调用 RegisterClassEx 注册的特定窗口类相关联。 CreateWindow 函数根据特定窗口类创建一个窗口。但基于一个窗口类，可以创建多个窗口。

窗口过程总是定义为如下形式:

```
LRESULT _export WndProc (HWND hwnd, UINT iMsg, UINT wParam, LONG lParam) ;
```

注意，窗口过程的四个参数与 MSG 结构的前四个域是相同的。

第一个参数 hwnd 是接收消息的窗口的句柄，它与 CreateWindow 函数的返回值相同。对于与 HELLOWIN 相似的程序 (只创建一个窗口)，这个参数是程序所知道的唯一窗口句柄。如果程序是基于同一窗口类 (同时也是同一窗口过程)创建多个窗口，则 hwnd 标识接收消息的特定窗口。

第二个参数是一个数值 (具体地讲，是一个 32 位的无符号整数，即 UINT)，它用以标识消息。最后两个参数 (一个叫做 wParam 的 WPARAM 和一个叫做 lParam 的 LPARAM) 提供有关消息的进一步信息，它们都叫“消息参数”。这些参数的内容与消息的类型相关。

2.2.16 处理消息

窗口过程所接受的每条消息均是用一个数值来标识的，也就是传给窗口过程的 iMsg 参数。Windows 头文件为每个消息参数定义以“WM”前缀打头的标识符。

一般来说，Windows 程序员使用 switch 和 case 结构来确定窗口过程接收的是什么消息以及如何适当地处理它。窗口过程在处理消息时，必须返回 0。窗口过程不予处理的所有消息应该被传给名为 DefWindowProc 的 Windows 函数。从 DefWindowProc 返回的值必须由窗口过程返回。

在 HELLOWIN 中，WndProc 只选择处理三种消息：WM_CREATE，WM_PAINT 和 WM_DESTROY。窗口过程的结构如下：

```
switch (iMsg)
     {
     case WM_CREATE:
          [处理 WM_CREATE 消息]
          return 0;
     case WM_PAINT:
          [处理 WM_PAINT 消息]
          return 0;
     case WM_DESTROY:
          [处理 WM_DESTROY 消息]
          return 0;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

调用 DefWindowProc 来为窗口过程不予处理的所有消息提供缺省处理，这是很重要的。

2.2.17 播放声音文件

窗口过程接收的第一个消息——也是 WndProc 选择处理的第一个消息——是 WM_CREATE。当 Windows 在 WinMain 中处理 CreateWindow 函数时，WndProc 接收这个消息。即，当 HELLOWIN 调用 CreateWindow 时，Windows 将做一些它必须做的工作。在这些工作中，Windows 调用 WndProc，将第一个参数设置为窗口句柄，第二个参数设置为 WM_CREATE。WndProc 处理 WM_CREATE 消息并将控制返回给 Windows。Windows 然后可以从 CreateWindow 调用中返回到 HELLOWIN 中，继续在 WinMain 中进行下一步的处理。

通常，窗口过程在 WM_CREATE 处理期间进行一次窗口初始化。HELLOWIN 对这条消息的处理中播放一个名为 HELLOWIN.WAV 的声音文件。它使用简单的 PlaySound 函数来做到这一点。第一个参数是文件的名字，也可以是在 Control Panel 的 Sound 中定义的一种声音的别名，或者是一个程序资源。第二个参数只有当声音文件是一种资源时才被使用。

第二个参数指定一些选项。在这个例子中，我指定第一个参数是一个文件名，并且异步地播放声音，即 PlaySound 函数调用在声音文件开始播放时立即返回，而不会等待它的完成。

WndProc 通过从窗口过程中返回 0，结束了整个 WM_CREATE 的处理。

2.2.18 WM_PAINT 消息

WndProc 处理的第二条消息为 WM_PAINT。这条消息在 Windows 程序设计中是很重要的。当窗口客户区域的一部分或者全部变为“无效”，以致于必须进行刷新时，将由这条消息通知程序。

客户区域怎么会变得无效呢？在最初创建窗口的时候，整个客户区域都是无效的，因为程序还没有在窗口上画什么东西。第一条 WM_PAINT 消息 (通常发生在 WinMain 中调用 UpdateWindow 时)指示窗口过程在客户区域上画一些东西。

在用户改变 HELLOWIN 窗口的大小后，客户区域重新变得无效。读者应该还记得，HELLOWIN 中 WndClassEx 结构的 style 参数设置为标志 CS_HREDRAW 和 CS_VREDRAW，这一风格指示 Windows，在窗口大小改变后，使整个窗口无效。然后，窗口过程将收到一条 WM_PAINT 消息。

当用户将 HELLOWIN 最小化，然后再次将窗口恢复为以前的大小时，Windows 将不会保存客户区域的内容。在图形环境下，这涉及的数据量很大。因此，Windows 令窗口无效，窗口过程接收一条 WM_PAINT 消息，并自动恢复其窗口的内容。

在移动窗口以致其相互重叠时，Windows 不保存一个窗口中被另一个窗口所遮盖的内容。在这一部分不再被遮盖之后，它就被标志为无效。窗口过程接收到一条 WM_PAINT 消息，以刷新窗口的内容。

对 WM_PAINT 的处理几乎总是从一个 BeginPaint 调用开始:

```
hdc = BeginPaint (hwnd, &ps) ;
```

而以一个 EndPaint 调用结束:

```
EndPaint (hwnd, &ps) ;
```

在这两个调用中，第一个参数都是程序的窗口句柄，第二个参数是指向类型为 PAINTSTRUCT 的结构指针。PAINTSTRUCT 结构中包含一些窗口过程，可以用来刷新客户区域的内容。我们将在下一章中讨论该结构的各个域。

在 BeginPaint 调用中，如果客户区域的背景还未被删除，则由 Windows 来删除。它使用注册窗口类的 WNDCLASSEX 结构的 hbrBackground 域中指定的刷子来删除背景。在 HELLOWIN 中，这是一个白色备用刷子。这意味着，Windows 将通过把窗口背景设置为白色来删除窗口背景。BeginPaint 调用令整个客户区域有效，并返回一个“设备环境句柄”。设备环境是指物理输出设备 (如视频显示器)及其设备驱动程序。在窗口的客户区域显示文本和图形需要设备环境句柄。但是不能用从 BeginPaint 返回的设备环境句柄在客户区域之外绘图，读者可以试一试。EndPaint 释放设备环境句柄，使之不再有效。

如果窗口过程不处理 WM_PAINT 消息 (这是很罕见的)，它们必须被传送给

DefWindowProc 。 DefWindowProc 只是依次调用 BeginPaint 和 EndPaint ，以使客户区域有效。

在调用完 BeginPaint 之后， WndProc 接着调用 GetClientRect:

```
GetClientRect (hwnd, &rect) ;
```

第一个参数是程序窗口的句柄。第二个参数是一个指针，指向一个名为 rect 的变量，该变量在 WndProc 中定义为 RECT 类型。

RECT 是一个"矩形"结构，在 WINDOWS.H 中定义。它有四个 int 域，分别为 left, top, right 和 bottom 。 GetClientRect 将这四个域设置为窗口客户区域的尺寸。 left 和 top 域通常设置为 0 ， right 和 bottom 域设置为客户区域的宽度和高度 (象素点数)。

WndProc 除了将该 RECT 结构指针作为 DrawText 的第四个参数传递外，不再对它做其它处理:

```
DrawText (hdc, "Hello, Window!", -1, &rect,
          DT_SINGLELINE|DT_CENTER|DT_VCENTER) ;
```

DrawText 可以输出文本 (正如其名字所表明的一样)。由于该函数要输出文本，第一个参数是从 BeginPaint 返回的设备环境句柄，第二个参数是要输出的文本，第三个参数是－1，指示文本串是以字节 0 终结的。

最后一个参数是一系列位标志，它们均在 WINDOWS.H 中定义。标志指示了文本必须显示在一行上，水平方向和垂直方向都位于第四个参数指定的矩形中央。因此，这个函数调用将导致串"Hello, Windows!"显示在客户区域的中央。

一旦客户区域变得无效 (正如在改变大小时所发生的情况一样)， WndProc 就接收一条新 WM_PAINT 消息。 WndProc 通过调用 GetClientRect 获取变化后的窗口大小，并在新窗口的中央显示文本。

2.2.19 WM_DESTROY 消息

WM_DESTROY 消息是另一条重要消息。这一条消息指示， Windows 正在根据用户输入的命令来清除窗口。该消息是用户在程序的系统菜单上选择 Close 或者按下 ALT-F4 时发生的。

HELLOWIN 通过调用 PostQuitMessage 以标准方式响应 WM_DESTROY 消息:

```
PostQuitMessage (0) ;
```

该函数在程序的消息队列中插入一条 WM_QUIT 消息。前面提到过， GetMessage 对于除了 WM_QUIT 之外的从消息队列中取出的所有消息都返回非 0 值。而当 GetMessage 取到一条 WM_QUIT 消息时，它返回 0 。这将导致 WinMain 退出消息循环，并终止程序。

2.3 Windows 程序设计的难点

即使有了对 HELLOWIN 的说明，读者对程序的结构和原理可能仍然觉得神秘。在为

传统环境编写简单的 C 程序时，整个程序可能包含在 main 函数中。而在 HELLOWIN 中，WinMain 只包含了注册窗口类，创建窗口，从消息队列中取出消息和发送消息所必须的代码。

程序的所有实际动作均在窗口过程中发生。在 HELLOWIN 中，这些动作不多，只是在窗口中显示一个文本串。但是在后面的章节中，读者将发现，Windows 程序所做的一切，都是响应发送给窗口过程的消息。这是概念上的主要难点之一，在开始编写 Windows 程序之前，必须先搞清楚。

2.3.1 别调用我，我会调用你

前面我们提到过，程序员已经熟悉了使用操作系统调用的思路。例如，C 程序员使用 open 或者 fopen 函数打开文件，编译程序提供的库函数最终通过调用操作系统来打开文件。这毫无问题。

但是 Windows 不同，尽管 Windows 有 1000 个以上的函数可供程序调用，但 Windows 也调用用户程序，比如前面定义的窗口过程 WndProc 。窗口过程与一个窗口类相关联，窗口类是程序调用 RegisterClass 注册的。基于该类创建的窗口使用这个窗口过程来处理窗口的所有消息。 Windows 通过调用窗口过程来给窗口发送一条消息。

在第一次创建窗口时， Windows 调用 WndProc 。在窗口清除时， Windows 也调用 WndProc 。窗口改变大小、移动或者变成图标时，从菜单中选择某一项、滚动滚动条或者按下鼠标按钮时，以及窗口客户区域必须被刷新时， Windows 都要调用 WndProc 。

所有这些调用都以消息的形式进行。在大多数 Windows 程序中，程序的主要部分都用来处理消息。 Windows 可以发送给窗口过程 200 多条不同的消息，它们都以 WM 打头的名字标识，并且都在 Windows 头文件中定义。

实际上，从程序外调用程序内的例程这一思想，在传统的程序设计中并非闻所未闻。C 中的 signal 函数可以捕获 Ctrl-Break 。读者可能有用汇编语言截获硬件中断的经验，也可能在 Microsoft BASIC 中使用过一种 ON 结构。 Microsoft 鼠标驱动程序提供了一种方法，使得非 Windows 程序可以用这种方法被告知鼠标的活动。

在 Windows 中，这种概念扩展为包括一切事件。窗口中发生的一切都以消息的形式传给窗口过程。然后，窗口过程以某种方式响应这条消息，或者将消息传给 DefWindowProc ，进行缺省处理。

在 HELLOWIN 中，窗口过程的 wParam 和 lParam 参数除了作为传递给 DefWindowProc 的参数外，不再有其它用处。这些参数给出了关于消息的其它信息，参数的含义与具体消息相关。

让我们来看一个例子。一旦窗口的客户区域大小发生了改变， Windows 就调用窗口的窗口过程。窗口过程的 hwnd 参数是改变大小的窗口的句柄。 message 参数为 WM_SIZE 。WM_SIZE 消息的 wParam 参数为常量 SIZENORMAL 、 SIZEICONIC 、 SIZEFULLSCREEN 、 SIZEZOOMSHOW 或 SIZEZOOMHIDE (在 WINDOWS.H 中定义为数值 0 到 4 之一)。 wParam 参数指示窗口是最大化或最小化，还是隐蔽（是另一个窗口最大化的结果)。 lParam 参数包含了新窗口的大小，新宽度和新高度均为 16 位值，合在一起

成为 32 位的 lParam 。 WINDOWS.H 中提供了帮助程序员从 lParam 中取出这两个值的宏，我们将在下一章中说明这个宏。

有时候， DefWindowProc 处理完消息后会产生其它的消息。例如，假设用户运行 HELLOWIN，并用键盘或者鼠标在系统菜单中选择 CLOSE 。 DefWindowProc 处理这一键盘或者鼠标输入，在检测到用户选择了 CLOSE 选项之后，它给窗口过程发送一条 WM_SYSCOMMAND 消息。 WndProc 将这条消息传给 DefWindowProc 。 DefWindowProc 给窗口过程发送一条 WM_CLOSE 消息来响应之。 WndProc 再次将它传给 DefWindowProc 。 DefWindowProc 调用 DestroyWindow 来响应这条 WM_CLOSE 消息。 DestroyWindow 导致 Windows 给窗口过程发送一条 WM_DESTROY 消息。 WndProc 再调用 PostQuitMessage，将一条 WM_QUIT 消息放入消息队列中，以此来响应 WM_DESTROY 消息。这条消息导致 WinMain 中的消息循环终止，然后程序结束。

2.3.2 进队消息与不进队消息

我们已经谈到过， Windows 给窗口发送消息，这意味着 Windows 调用窗口过程。但是， Windows 程序也有一个消息循环，它调用 GetMessage 从消息队列中取出消息，并且调用 DispatchMessage 将消息发送给窗口过程。

那么， Windows 程序是依次等待消息（类似于普通程序中等价的键盘输入），然后将消息送到某地方去的吗？或者，它是直接从程序外面接收消息的吗？实际上，两种情况都存在。

消息可以分为“进队的”和“不进队的”。进队的消息是由 Windows 放入程序消息队列中的，并且在消息队列中取出和发送。不进队的消息在 Windows 调用窗口过程时直接送给窗口。结果是窗口过程获取窗口的所有消息，包括进队的和不进队的。从结构上看，Windows 程序很清晰，因为它有一个消息处理中心，进队消息“邮递”给消息队列，而不进队消息“发送”给窗口过程。

进队消息基本上是用户输入的结果，以击键（如 WM_KEYDOWN 和 WM_KEYUP)、击键产生的字符（如 WM_CHAR)、鼠标移动 (WM_MOUSEMOVE) 和鼠标按钮 (WM_LBUTTONDOWN)的形式给出。进队消息还包含时钟消息 (WM_TIMER)、刷新消息 (WM_PAINT)和退出消息 (WM_QUIT)。不进队消息则是其它消息。在许多情况下，不进队消息来自进队消息。在窗口过程内将一个不进队消息传给 DefWindowProc 时， Windows 通常通过给窗口过程发送其它消息来处理这条消息。

这一过程显然很复杂，但幸运的是，其中的大部分是由 Windows 解决的，不关我们的程序的事。从窗口过程的角度来看，这些消息是以一种有序的、同步的方式进出的。窗口过程可以处理它们，也可以不处理。由于这个原因，窗口过程被称做“最后的陷阱”。消息几乎将所有影响窗口的事件告知窗口过程。

不进队消息通常是调用某个 Windows 函数或者调用 SendMessage 显式发送一条消息的结果（消息也可以通过调用 PostMessage 来放入消息队列)。

例如，在 WinMain 调用 CreateWindow 时， Windows 创建窗口，并在此过程中给窗口发送一条 WM_CREATE 消息。在 WinMain 调用 ShowWindow 时， Windows 给窗口发送

WM_SIZE 和 WM_SHOWWINDOW 消息。在 WinMain 调用 UpdateWindow 时， Windows 给窗口过程发送一条 WM_PAINT 消息。

消息不同于硬件中断。在窗口过程中处理一条消息时，程序不会被另一条消息所中断。只有在窗口过程调用一个产生新消息的函数时，窗口过程才会在函数返回前处理这条新消息。

消息循环与窗口过程不是并发运行的。窗口过程处理进队消息是在 WinMain 中调用 DispatchMessage 的结果。直到窗口过程处理完消息之后， DispatchMessage 才会返回。

但是请注意，窗口过程必须是可重入的。也就是说，在 WndProc 为了处理一条消息而调用 DefWindowProc 时， Windows 经常产生一条新的消息又来调用 WndProc 。这是 Windows 程序需要 8KB 堆栈的理由之一；可以在模块定义文件 (.DEF)中定义栈的大小。在大多数情况下，窗口过程的可重入性不会带来任何问题，但是程序员应该意识到这一点。

例如，假设你在窗口过程中处理一条消息时设置了一个变量，然后调用了一个 Windows 函数。在这个函数返回时，你还能保证那个变量的值还是原来那个吗？难说——很可能你调用的 Windows 函数产生了另外一条消息，并且窗口过程在处理这条消息时改变了该变量的值。这也是在编译 Windows 程序时，有些编译优化选项必须关闭的原因之一。

在许多情况下，窗口过程必须保存它从消息中获取的信息，并在处理另一条消息时使用这些信息。这些信息可以存储在窗口的 Static 变量或全程变量中。

当然，读者将在下面几章对此有一个更清楚的理解，因为窗口过程将处理更多的消息。

2.3.3　别像只懒猪

Windows 95 是一种抢占式的多任务环境。这意味着当一个程序在进行一项长时间工作时， Windows 可以允许用户将控制切换到另一个程序中。这是一件好事，也是 Windows 95 优越于以前基于 DOS 的 Windows 版本的地方。

然而，由于 Windows 构造的方式，这种抢占式多任务并不总是以你希望的方式工作。例如，假设你的程序花费一分钟左右来处理某一个消息。是的，用户可以将控制切换到另一个程序，但是却无法对你的程序进行任何动作。用户无法移动你的程序窗口，缩放它，关闭它，什么都不能做。这是因为必须由你的窗口过程来执行这些函数，而你的窗口过程正在忙于作一项长期的工作。表面上并不是窗口过程在执行它自己的移动和缩放操作，但实际上确实是它在做。这就是 DefWindowProc 部分的工作，它必须被考虑为你的窗口过程的一部分。

如果你的程序在处理某些消息时需要长时间工作的话，可以选择我在第十四章里描述的那些方法来做得更有礼貌一些。即使是在抢占式多任务环境中，也不应该让你的程序呆在屏幕上一动不动。这会让用户讨厌的，他们会认为你的程序像只猪一样。

2.3.4　学习曲线

读者毫无疑问已经从本章得出一条结论： Windows程序设计与传统环境 (如MS-DOS) 下的程序设计大相径庭。不会有人声称 Windows 程序设计很容易了。

我刚开始学习 Windows 程序设计时，决定如以前那样，在学习新操作系统或者新语言时，编写一个简单的“打印十六进制数”程序来显示文件内容。在传统的 MS-DOS 环境下，这样的程序涉及到命令行处理、基本的文件 I/O 和屏幕输出格式。但是，这次的 Windows 十六进制打印程序变成了一个“怪物”。它要求学习菜单、对话框、滚动条等等。把它作为第一个 Windows 程序，这实在是一个错误，因为要求学习的东西一下子太多了。

在这个程序编写完之后，它与我先前编写的十六进制打印程序完全不同。WINDUMP (我这样叫它)不是从命令行中获取文件名，而是提供一个列表框，列出当前目录中的所有文件。WINDUMP 不是以简单的电传打字机的方式将输出送到屏幕上，而是具有滚动条，可以移动到文件的任意位置。另一个特殊的优点是：可以运行 WINDUMP 的两个副本，以逐行比较两个文件。简言之，WINDUMP 是我编写的十六进制打印程序中第一次感到自豪的程序。

读者应当考虑一下：想不想编写出新颖的、具有创意的用户界面程序 (界面中包括菜单、对话框、滚动条和图形)？如果想，接下来的问题是：是自己编写所有菜单、对话框、滚动条和图形代码，还是利用 Windows 内已经有的东西呢？换句话说，是学习使用 1000 种函数调用容易呢，还是自己编写更容易？是使用消息驱动式体系结构编程容易呢，还是坚持使用传统模式下的几种不同输入源容易？

如果读者决心自己来编写用户界面，那么最好放下这本书，立即投入工作。乐意使用 Windows 来编程的读者，请继续学习在窗口中如何显示和滚动文本。

第3章 输出文本

在前一章中，你看到了一个简单的 Windows 程序，它在窗口中央，或者更准确地说，在客户区中央显示一行文本。应用程序窗口和它的客户区之间的区别是非常重要的：客户区是指窗口中未被标题条、窗口边框、菜单条和滚动条占据的地方。客户区是窗口中可以由程序任意书写的部分。

对于客户区，你几乎可以为所欲为，只不过必须认为其大小将是某一特定尺寸，或者在程序运行时其大小保持不变。如果你熟悉了 IBM PC 编程，这些例外可能会使你感到惊讶：你不能再考虑 25 行（或 43 行或 50 行） 80 列的屏幕了。你的程序必须与其它 Windows 程序共享视频显示器。 Windows 程序的设计者要控制程序窗口如何在屏幕上显示。尽管可以创建一个特定的固定大小的窗口（这对于计算器之类的应用是合理的），但在大多数情况下，窗口的大小应该是可变的。你的程序必须能够接受指定给它的大小，并且合理地利用这一空间。

这有两种可能的情况。一种可能是，程序只有仅能显示“ hello ”的客户区；还有一种可能是，程序在一个大屏幕、高分辨率的系统上运行，其客户区大得足以显示两整页文本。灵活地处理这两种极端是 Windows 程序设计的要点之一。

尽管 Windows 为显示图形提供了强大的图形设备接口(GDI)函数，但在这一章中，我将暂时集中介绍文本的显示。我也将忽略不同字体（外形）及字体大小，使用 Windows 的缺省系统字体——虽然 Windows 提供了很多字体和字号。这看起来似乎是一种限制，其实不然，本章涉及和解决的问题适用于所有 Windows 程序设计。在显示文本和图形（例如 Windows 的 CALENDAR 、 CARDFILE 、 CALCULATOR 等程序）时， Windows 缺省系统字体的字符大小通常决定了图形的尺寸。

本章表面上是讨论如何绘图，实际上是讨论与设备无关的程序设计基础。 Windows 程序只能对客户区大小甚至字符的大小做很少的假定，相反，必须使用 Windows 提供的功能来获取关于环境的信息.

3.1 绘图和刷新

在 MS-DOS 下，以全屏幕方式使用显示器的程序可以在显示器的任意部分上输出，程序输出到屏幕上的内容会停留在原处，不会神秘地消失。因此，程序可以丢掉重新生成屏幕显示时所需的信息。如果是另一个程序(如某个 RAM 驻留弹出程序)覆盖了显示器的部分表面，则该程序负责在退出时恢复显示的内容.

在 Windows 中，只能在用户窗口的客户区进行显示，而且不能确保在客户区内显示的

内容会一直保留到程序有意地改写它。例如，另一个应用程序的对话框可能覆盖客户区的一部分。尽管 Windows 试图保存并恢复被对话框覆盖的显示区域，但有时候却做不到这一点。对话框从屏幕上擦除后，Windows 将要求用户程序刷新客户区的这一部分。

Windows 是一个消息驱动系统。它通过发送应用程序消息队列中的消息或者把消息发送给合适的窗口过程，将发生的各种事件通知给应用程序。如果要将窗口的部分客户区需要更新的事件通知窗口过程，则可以发送 WM_PAINT 消息。

3.1.1 WM_PAINT 消息

大多数 Windows 程序在进入消息循环之前的初始化期间都要调用函数 UpdateWindow，Windows 利用这个机会给窗口过程发送第一条 WM_PAINT 消息。这条消息通知窗口过程：绘制客户区的准备工作已经就绪。此后，窗口过程应在任何时刻都准备好处理其他 WM_PAINT 消息，必要的话，甚至刷新窗口的整个客户区。在下面几种情况之一发生时，窗口过程将接收到一条 WM_PAINT 消息:

- 在用户移动窗口或者显示窗口时，窗口中先前被隐藏的区域重新可见。
- 用户改变窗口的大小(如果窗口类风格 CS_HREDRAW 和 CS_VREDRAW 置位)。
- 程序使用 ScrollWindow 或者 ScrollDC 函数滚动客户区。
- 程序使用 InvalidateRect 或者 InvalidateRgn 函数显式产生一条 WM_PAINT 消息。

在某些情况下，客户区的一部分被临时覆盖，Windows 试图保存一个显示区域，并在以后恢复它，这并不总是能成功的。下面的情况下，Windows 可能发送一条 WM_PAINT 消息:

- Windows 擦除对话框或者消息框，它们覆盖了部分窗口。
- 菜单下拉出来，然后被释放。

在某些情况下，Windows 总是保存它所覆盖的显示区域，然后恢复它。这些情况是:

- 光标穿越客户区。
- 图标拖过客户区。

处理 WM_PAINT 消息要求程序员改变自己向显示器输出的思维方式。程序应该组织得可以保留绘制客户区需要的所有信息，并且仅当“应要求”——即 Windows 给窗口过程发送一条 WM_PAINT 消息时才进行刷新。如果程序需要刷新其客户区，它可以强制 Windows 产生一条 WM_PAINT 消息。这看来似乎是在屏幕上显示内容的一种舍近求远的方法。但你的程序结构可以从中受益。

3.1.2 有效矩形和无效矩形

尽管窗口过程一旦接收到 WM_PAINT 消息之后，就准备更新整个客户区，但它经常只需要刷新一个较小的区域(最常见的是矩形区)。显然，当对话框覆盖了部分客户区时，情况即是如此。在擦除对话框之后，需要刷新的只是先前被对话框遮住的矩形区域。

这个区域称为“无效区域”或者叫“更新区域”。正是客户区内无效区域的存在，才提示了 Windows 将一条 WM_PAINT 消息放在应用程序的消息队列中。只有在客户区的某

一部分失效时，窗口过程才会接收 WM_PAINT 消息。

Windows 内部为每个窗口保存一个“绘图信息结构”，这个结构包含了包围无效区域的最小矩形的坐标以及其它信息，这个矩形就叫做“无效矩形”，有时也称为“无效区域”。如果在窗口过程处理 WM_PAINT 消息之前客户区中的另一个区域变为无效，则 Windows 计算出一个新的无效区域(该区域包围了两个区域)，并将这种变化后的信息放在绘图信息结构中。 Windows 不会将多条 WM_PAINT 消息都放在消息队列中。

窗口过程可以通过调用 InvalidateRect 使客户区内的一个矩形无效。如果消息队列中已经包含一条 WM_PAINT 消息， Windows 将计算出一个新的无效矩形。否则，它将一条新的 WM_PAINT 消息放入消息队列中。在接收到 WM_PAINT 消息时，窗口过程可以获取无效矩形的坐标(我们马上就会看到这一点)。通过调用 GetUpdateRect，可以在任何时候获取这些坐标。

在处理 WM_PAINT 消息期间，窗口过程在调用了 BeginPaint 之后，整个客户区即变为有效。程序也可以通过调用 ValidateRect 函数使客户区内的任意矩形区域变为有效。如果这条调用具有令整个无效区域变为有效的效果，则当前队列中的任意 WM_PAINT 消息都将被删除。

3.2 GDI 简 介

要在窗口的客户区绘图，可以使用 Windows 的图形设备接口(GDI)函数。在第十一至第十五章有对 GDI 的完整讨论。 Windows 提供了五个 GDI 函数，用于将文本串输出到窗口的客户区内。我们已经在第一章遇到过 DrawText 函数，但是目前使用最为普遍的文本输出函数是 TextOut 。该函数的格式如下:

```
TextOut (hdc, x, y, lpsString, nLength) ;
```

TextOut 向显示器上输出一个字符串。 lpsString 参数是一个指向字符串的长指针，nLength 是串的长度。 x 和 y 参数定义了字符串在客户区的开始位置(使用逻辑坐标)。 hdc 参数是一个“设备环境句柄”，它是 GDI 的重要部分。实际上，每个 GDI 函数都需要将这个句柄作为函数的第一个参数。

3.2.1 设备环境

读者可能还记得，句柄只不过是一个数值， Windows 用它在内部引用一个对象。程序员从 Windows 获取句柄，然后在其它函数中使用该句柄。设备环境句柄是 GDI 函数的窗口“口令”，有了这种设备环境句柄，程序员就能自如地在客户区上绘图，使图形如自己所愿地变得好看或者难看。

设备环境（简称为“ DC ”)实际上是 GDI 保存的一个数据结构。设备环境与特定的显示设备(如打印机、绘图仪或者视频显示器)相关。对于视频显示器，设备环境总是与显示器上的特定窗口相关。

设备环境中的有些值是图形“属性”，这些属性定义 GDI 绘图函数工作的细节。例如，对于 TextOut，设备环境的属性确定了文本的颜色、文本的背景色、x 坐标和 y 坐标如何映射到窗口的客户区以及在显示文本时 Windows 使用的字体。

当程序需要绘图时，必须先获取设备环境句柄。在绘图完毕后，程序必须释放该句柄。句柄被程序释放后就不再有效，且不能再被使用。程序必须在处理单个消息期间获取和释放句柄。除了调用 CreateDC 创建的设备环境之外，程序不能在两个消息之间保存其它设备环境句柄。

Windows 应用程序一般使用两种方法来获取设备环境句柄，以备在屏幕上绘图。

3.2.2 获取设备环境句柄：方法一

在处理 WM_PAINT 消息时，使用这种方法。它涉及 BeginPaint 和 EndPaint 两个函数，这两个函数要求窗口句柄（作为参数传给窗口过程）和一个 PAINTSTRUCT 的结构变量的地址为参数。Windows 程序员通常把这一结构变量命名为 ps 并且在窗口过程中定义它：

```
PAINTSTRUCT ps ;
```

在处理 WM_PAINT 消息时，窗口过程首先调用 BeginPaint 来填入 ps 结构的域。BeginPaint 返回的值是设备环境句柄，这一返回值通常被保存在一个叫做 hdc 的变量中。它在窗口过程中的定义如下：

```
HDC hdc ;
```

HDC 数据类型在 WINDOWS.H 中定义为 HANDLE，这是一个 32 位的无符号整数。然后，程序就可以使用 TextOut 等 GDI 函数。调用 EndPaint 即可以释放设备环境句柄。

一般地，处理 WM_PAINT 消息的形式如下:

```
case WM_PAINT:
     hdc = BeginPaint(hwnd, &ps) ;
     [使用 GDI 函数]
     EndPaint(hwnd, &ps) ;
     return 0;
```

在处理 WM_PAINT 消息时，必须成对地调用 BeginPaint 和 EndPaint。如果窗口过程不处理 WM_PAINT 消息，则它必须将 WM_PAINT 消息传递给 Windows 中 DefWindowProc(缺省窗口过程)。

DefWindowProc 以下列代码处理 WM_PAINT 消息:

```
case WM_PAINT:
     BinPaint(hwnd, &ps) ;
     EndPaint(hwnd, &ps) ;
     return 0 ;
```

这一 BeginPaint 和 EndPaint 序列中间没有任何语句，仅仅使先前无效区域变为有效。但以下方法是错误的:

```
case WM_PAINT:
    return 0 ;    //错误!!!
```

Windows 将一条 WM_PAINT 消息放到消息队列中，是因为客户区的一部分无效。如果不调用 BeginPaint 和 EndPaint(或者 ValidateRect)，则 Windows 不会使该区域变为有效。相反，Windows 将发送另一条 WM_PAINT 消息，且一直发送下去。

3.2.3 绘图信息结构

前面提到过，Windows 为每个窗口保存一个“绘图信息结构”，这就是 PAINTSTRUCT 。它在 WINDOWS.H 中定义如下:

```
typedef struct tagPAINTSTRUCT
    {
    HDC   hdc ;
    BOOL  fErase ;
    RECT  rcPaint ;
    BOOL  fRestore ;
    BOOL  fIncUpdate ;
    BYTE  rgbReserved[16] ;
    }
    PAINTSTRUCT ;
```

在程序调用 BeginPaint 时，Windows 填充该结构的各个域。用户程序只使用前三个域，其它域由 Windows 内部使用。

hdc 域是设备环境句柄。在旧版本的 Windows 中，BeginPaint 的返回值也曾是这个设备环境句柄。

在大多数情况下，fErase 被标志为 TRUE(非零)，这意味着 Windows 已经擦除了无效矩形的背景。Windows 使用 WNDCLASSEX 结构的 hbrBackground 域指定的刷子来擦除背景，这个 WNDCLASSEX 结构是程序在 WinMain 初始化期间登录窗口类时使用的。许多 Windows 程序使用白色刷子:

```
wndclass.hbrBackgroup = (HBRUSH) GetStockObject(WHITE_BRUSH) ;
```

不过，如果程序通过调用 Windows 函数 InvalidateRect 使客户区中的一个矩形失效，则该函数的某个参数会指定是否擦除背景。如果这个参数为 FALSE(即 0)，则 Windows 将不会擦除背景，且 fErase 域也将为 FALSE 。

PAINTSTRUCT 结构的 rcPaint 域是 RECT 类型的结构。你已经在第二章中看到，RECT 结构定义了一个矩形，其四个域为 left 、 top 、 right 和 bottom 。 PAINTSTRUCT 结构的 rcPaint 域定义了无效矩形的边界，如图 3-1 所示。这些值均以象素点为单位，并相对于客户区的左上角。无效矩形是程序需要刷新的区域。尽管 Windows 一旦接收到 WM_PAINT 消息后就能刷新窗口的整个客户区，但只刷新该矩形所定义的窗口区域能节省时间。

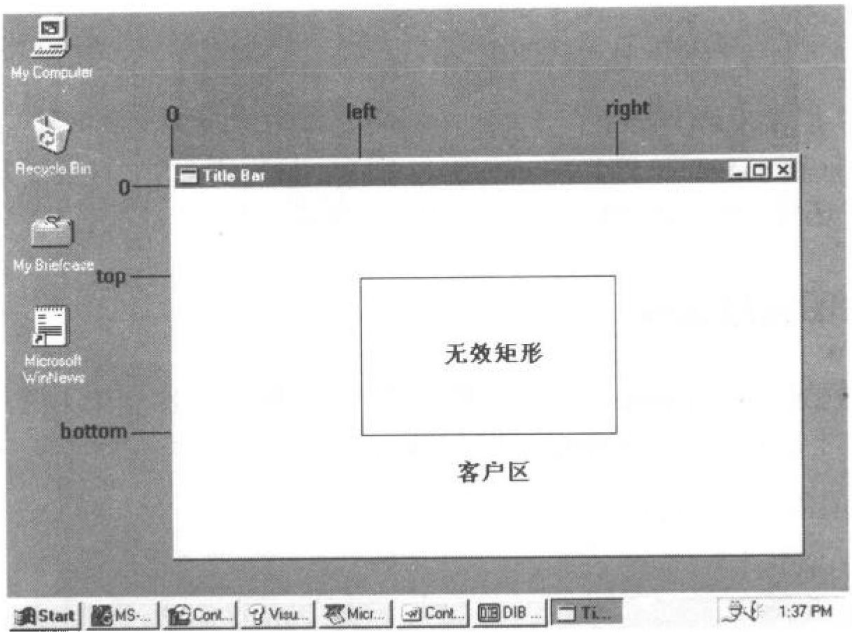


图 3-1 无效矩形的边界

PAINTSTRUCT 中的 rcPaint 矩形不仅是无效矩形，它还是一个“剪取”矩形。这意味着 Windows 将绘图操作限制在剪取矩形内(更确切地说，如果无效矩形区域不为矩形，则 Windows 将绘图操作限制在这个区域内)。在使用 PAINTSTRUCT 结构中的设备环境句柄时， Windows 将不会在 rcPaint 矩形之外绘图。

在处理 WM_PAINT 消息时，为了在 rcPaint 矩形外绘图，可以使用如下调用:

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

该调用在 BeginPaint 调用之前进行，它使整个客户区变为无效，并擦除背景。但是，如果最后一个参数等于 FALSE，则不擦除背景，原有的东西将保留在原处。

在第二章的 HELLOWIN 程序中，我们并不关心处理 WM_PAINT 消息时的无效矩形或者剪取矩形。如果文本显示区域恰巧在无效矩形内，则由 DrawText 恢复之。否则，在处理 DrawText 调用的某个时刻， Windows 会确定它无须向显示器上输出。不过，这一决定需要时间。关心程序性能和速度的程序员希望在处理 WM_PAINT 期间使用无效矩形范围，以避免不必要的 GDI 调用。

3.2.4 获取设备环境句柄：方法二

如果你想在处理非 WM_PAINT 的消息时绘制客户区，或者需要将设备环境句柄用于其它目的，如获取关于设备环境的信息，你也能获取设备环境句柄。这时，调用 GetDC 获取设备环境句柄，并在使用它之后调用 ReleaseDC:

```
hdc = GetDC(hwnd) ;
[使用 GDI 函数]
ReleaseDC(hwnd, hdc) ;
```

与 BeginPaint 和 EndPaint 一样， GetDC 和 ReleaseDC 函数必须成对地使用。如果在处理某消息时调用 GetDC ，则必须在退出窗口过程之前调用 ReleaseDC 。不要调用 GetDC 响应一条消息，又调用 ReleaseDC 响应另一条消息。

与 PAINTSTRUCT 结构中获取的设备环境句柄不同， GetDC 返回的设备环境句柄具有一个剪取矩形，它等于整个客户区。可以在客户区的某一部分绘图，而不只是在无效矩形上绘图(如果确实存在无效矩形)。与 BeginPaint 不同， GetDC 不会使任何无效区域变为有效。

一般可以调用 GetDC 和 ReleaseDC 来对键盘消息（如在字处理程序中）和鼠标消息（如在画图程序中）作出反应。此时，程序可以立刻根据用户的键盘或鼠标输入来更新客户区，而不需要考虑为了窗口的无效区域而使用 WM_PAINT 消息。不过，一旦确实收到了 WM_PAINT 消息，程序就必须要收集足够的信息后才能更新显示。

3.2.5 TextOut：细节

在获取设备环境句柄时， Windows 用缺省值填充设备环境结构。在后面几章将会看到，你可以用 GDI 函数改变这些缺省值。目前，我们所感兴趣的 GDI 函数是 TextOut:

```
TextOut(hdc, x, y, lpsString, nLength) ;
```

以下将详细地讨论这个函数。

第一个参数是设备环境句柄，它既可以是 GetDC 的返回值，也可以是在处理 WM_PAINT 消息时 BeginPaint 的返回值。

设备环境的属性控制了被显示的文本串的特征。例如，设备环境中有一个属性指定文本颜色，缺省颜色为黑色；缺省设备环境还定义了白色的背景。在程序向显示器输出文本时， Windows 使用这个背景色来填充字符周围的空间。

该文本背景色与定义窗口类时设置的背景并不是同一个东西。窗口类中的背景是一个刷子，它是一种纯色或者非纯色组成的刷子， Windows 用它来擦除客户区，它不是设备环境结构的一部分。在定义窗口类结构时，大多数 Windows 应用程序使用 WHITE_BRUSH ，以便缺省设备环境中的背景颜色与 Windows 用以擦除客户区背景的颜色相同。

lpsString 参数是指向字符串的长指针， nLength 是串的长度。串中不能包含任何 ASCII 控制字符(如回车、换行、制表或退格)， Windows 会将这些控制字符显示为实心块。TextOut 不识别串尾标志 0 ，而需要 nLength 参数指明长度。

TextOut 中的 x 和 y 定义客户区内字符串的开始位置， x 是水平位置； y 是垂直位置。串中第一个字符的左上角位于坐标点(x ， y)。在缺省设备环境中，原点（ x 和 y 均为 0 的点)是客户区的左上角。如果在 TextOut 中将 x 和 y 置为 0 ，则将从客户区左上角开始输出字符串。

GDI 坐标是“逻辑坐标”。 Windows 有许多“映射方式”，它们用来控制 GDI 函数中指定的逻辑坐标如何转换为显示器的物理象素坐标。映射方式在设备环境中定义，缺省映射方式是 MM_TEXT(使用 WINDOWS.H 中的标识符)。在 MM_TEXT 映射方式下，逻辑单位与物理单位相同，都是象素点； x 的值从左向右递增， y 的值从上到下递增(参看图

3-2)。MM_TEXT 坐标系与 Windows 在 PAINTSTRUCT 结构中定义无效矩形时使用的坐标系相同，这为我们带来了很多方便(但是，其它映射方式并非如此)。

设备环境也定义了一个剪取区域。你已经看到，对于从 GetDC 获取的设备环境句柄，缺省剪取区域是整个客户区；而对于从 BeginPaint 获取的设备环境句柄，则为无效区域。Windows 将不会在剪取区域之外的任意位置显示字符串。如果一个字符有一部分在剪取区域外，则 Windows 将只显示此区域内的那部分。要想将输出写到窗口的客户区之外不是那么容易的，所以不用担心会无意识地这样做。

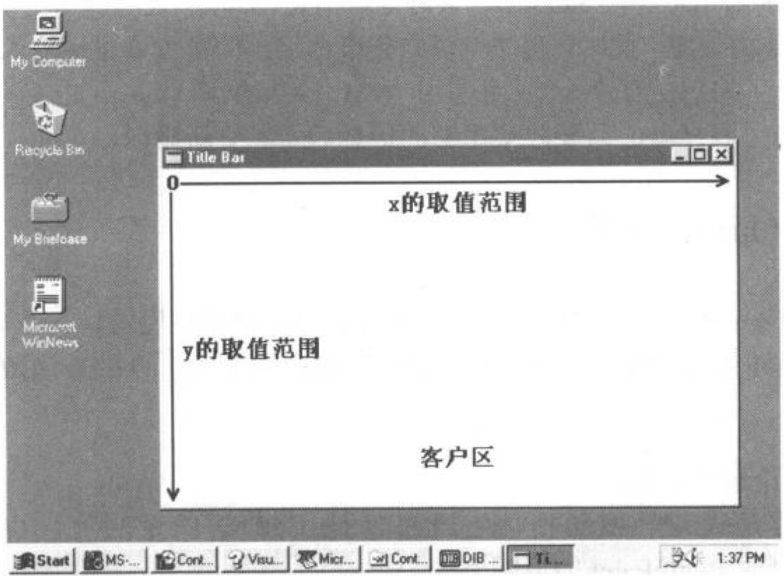


图 3-2 MM_TEXT映射方式下的x坐标和y坐标

3.2.6 系统字体

设备环境还定义了 Windows 在将文本输出到客户区时使用的字体。缺省字体为“系统字体”，或用 Windows 头文件中的标识符即 SYSTEM_FONT 。系统字体是 Windows 用来在标题条、菜单和对话框中显示文本的字体。

在 Windows 的早期版本中，系统字体是等大(fixed-pitch)字体，这意味着所有字符均具有同样的宽度，非常类似于一个打字机。然而，从 Windows 3.0 开始(并一直持续到 Windows 95)，系统字体成为一种可变(variable-pitch)字体，这意味着不同的字符具有不同的大小，比如，“ W ”要比“ I ”宽。可变字体比等大字体要好读，这已经是一个公认的事实。不过，可以想见，这一转变使很多原来的 Windows 程序代码不再适用，从而要求程序员学习一些使用字体的新技术。

系统字体是一种“光栅字体”，这意味着字符被定义为象素块。 Windows 的零售版本中包含有好几种不同大小的系统字体，以用于不同的视频显示器。当新视频卡的制造商开发出一种新的 Windows 显示驱动器之后，他们也负责开发一种适合于该显示器分辨率的新系统字体；或者，制造商会指定使用现有 Windows 版本中的系统字体文件。系统字体必须设计为至少能在显示器上显示 25 行 80 列，这是你能得到的关于屏幕大小与 Windows 字体

大小关系的唯一承诺。

3.2.7 字符大小

要用 TextOut 显示多行文本，就必须确定字体的字符大小，可以根据字符的高度来定位字符的后续行，以及根据字符的宽度来定位字符的后续列。

调用 GetTextMetrics 就可以获取字符的大小。由于它返回设备环境中的字体信息，所以 GetTextMetrics 需要一个设备环境句柄。Windows 将文本尺寸的不同值拷贝到 TEXTMETRICS 类型的结构中，值的单位取决于选定的设备环境映射方式。在缺省设备环境下，这种映射方式为 MM_TEXT，所以值的大小是以象素点为单位的。

要使用 GetTextMetrics 函数，需要先定义一个结构变量（通常称为 tm）：

```
TEXTMETRICS tm ;
```

然后获取设备环境句柄，并调用 GetTextMetrics:

```
hdc = GetDC(hwnd) ;
GetTextMetrics(hdc, &tm) ;
ReleaseDC(hwnd, hdc) ;
```

此后，你就可以查看文本尺寸结构中的值，并有可能保存其中的一些以备将来使用。

3.2.8 文本尺寸：细节

TEXTMETRIC 结构提供了关于当前设备环境中选用的字体的丰富信息。但是，字体的纵向尺寸只由 5 个值确定，如图 3-3 所示。

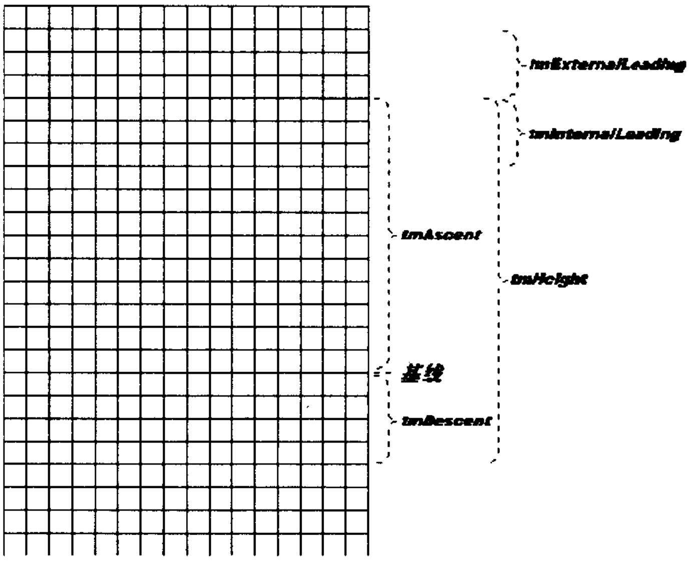


图 3-3 定义字体中纵向字符大小的五个值

这些值的意思基本上是不言自明的。值 tmInternalLeading 是为字符上的着重号所预留的空间大小。如果该值设置为 0，则加着重号的大写字母被缩短，以适合于字符的加重。值 tmExternalLeading 是字体设计者建议加在横向字符之间的空间大小。在安排文本行之间的空隙时，你可以接受设计者建议的值，也可以拒绝之。

TEXTMETRICS 结构包含有描述字符宽度的两个域，即 tmAveCharWidth(小写字母加权平均宽度)和 tmMaxCharWidth(字体中最宽字符的宽度)。对于固定间距的字体，这两个值是相等的。

本章的示例程序还需要另一种字符宽度，即大写字母的平均宽度，这可以用 tmAveCharWidth 乘以 150%大致计算出来。

必须认识到，系统字体的尺寸取决于 Windows 所运行的视频显示器的分辨率。Windows 提供了一个与设备无关的图形界面，但程序员并不是就此无所事事。不要想当然地猜测字符尺寸来编写 Windows 程序，也不要把值定死，可以使用 GetTextMetrics 函数获取这一信息。

3.2.9 格式化文本

由于在一次 Windows 对话中间，系统字体的尺寸不会发生改变，因此在程序运行过程中，程序员只需要调用一次 GetTextMetrics 。最好是在窗口过程中处理 WM_CREATE 消息时进行此调用， WM_CREATE 消息是窗口过程接收的第一条消息。当在 WinMain 中调用 CreateWindow 时， Windows 以一条 WM_CREATE 消息调用窗口过程。

假设要编写一个 Windows 程序，在客户区显示几行文本，这需要先获取字符宽度和高度。你可以在窗口过程内定义两个变量来保存平均字符宽度和高度:

```
static short cxChar, cyChar ;
```

变量名的前缀 c 代表“ count ”，与 x 和 y 结合，分别指宽和高。这些变量定义为 static ，因为它们在窗口过程处理其它消息(如 WM_PAINT)时也应该是有效的。如果变量在函数外面定义，则不需要定义为 static 。

下面是 WM_CREATE 代码:

```
case WM_CREATE:
     hdc = GetDC(hwnd) ;

     GetTextMetrics(hdc, &tm) ;
     cxChar = tm.tmAveCharWidth ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     ReleaseDC(hwnd, hdc) ;
     return 0 ;
```

如果文本行距不想包括外部间距，则可以使用:

```
cyChar = tm.tmHeight ;
```

如何使用这些字符尺寸来计算显示坐标取决与程序员。简单的方法是在客户区顶部留一个 cyChar 边界，在左边留一个 cxChar 边界。要显示多行左对齐的文本，可以在调用 TextOut 函数时分别使用如下的 x 坐标值和 y 坐标值:

```
cxChar
cyChar * (1+i)
```

其中，i 是从 0 开始的行数。

程序员常常需要显示格式化的数字和简单的字符串。如果是在 MS-DOS 中用标准 C 库函数编程，则可以使用 printf 。但在 Windows 下则不能使用 printf，因为 printf 输出到标准输出设备，而 Windows 中无此概念。

与此相对应，我们可以使用 sprintf 。 sprintf 函数与 printf 相似，只是 sprintf 将格式化串放到字符数组中，然后，可以用 TextOut 将串输出到显示器上。而且， sprintf 的返回值就是串的长度，程序员可以将这个值传递给 TextOut 作为 nLength 参数。下面的代码显示了一种 sprintf 与 TextOut 的典型组合:

```
int iLength;
char szBuffer[40];
[其它程序行]
iLength = sprintf(szBuffer, "The sum of %d and %d is %d", nA, nB, nA + nB) ;
TextOut(hdc, x, y, szBuffcr, iLength) ;
```

对于这样简单的情况，可以将 nLength 的赋值与 TextOut 放在同一条语句中，从而无需定义 iLength:

```
TextOut(hdc, x, y, szBuffer,
   sprintf(szBuffer, "The sum of %d and %d is %d", nA, nB, nA+nB) ;
```

虽然风格欠佳，但是功能与前者是一样的。

如果不需要显示浮点数，可以使用 wsprintf 来代替 sprintf 。 wsprintf 函数与 sprintf 语法相同，但前者是 Windows 提供的，所以使用它不会增大.EXE 文件。

3.2.10 综合使用

现在，我们似乎已经具备了在屏幕上显示多行文本所需要的所有知识。我们知道如何获取一个设备环境句柄，如何使用 TextOut 函数以及如何根据字符大小来安排字距，剩下的就是显示一点有意义的东西了。

从 Windows 的 GetSystemMetrics 中获取的信息是很有意义的，该函数返回 Windows 中不同图形构件的大小信息。图形构件包括图标、光标、标题条和滚动条等，它们的大小因显示适配器和驱动器的不同而有所不同。 GetSystemMetrics 需要一个参数，叫做“索引”，索引是 WINDOWS.H 中定义的 73 个整数标识符之一。 GetSystemMetrics 返回一个整数，这个整数通常是参数中指定的构件大小。

让我们来编写一个程序，显示一些可以从 GetSystemMetrics 调用中获取的信息，显示格式为每种构件一行。如果我们建立一个头文件，在头文件中定义一个结构数组，此结构

包含 GetSystemMetrics 索引对应的 WINDOWS.H 标识符和调用所返回的每个值对应的字符串，这样处理起来要容易一些。头文件名为 SYSMETS.H，如图 3-4 所示。

SYSMETS.H

```
/*-----------------------------------------------
   SYSMETS.H -- System metrics display structure
  -----------------------------------------------*/

#define NUMLINES ((int) (sizeof sysmetrics / sizeof sysmetrics [0]))

struct
{
int  iIndex ;
char *szLabel ;
char *szDesc ;
}
sysmetrics [] =
{
SM_CXSCREEN,          "SM_CXSCREEN",          "Screen width in pixels",
SM_CYSCREEN,          "SM_CYSCREEN",          "Screen height in pixels",
SM_CXVSCROLL,         "SM_CXVSCROLL",         "Vertical scroll arrow width",
SM_CYHSCROLL,         "SM_CYHSCROLL",         "Horizontal scroll arrow height",
SM_CYCAPTION,         "SM_CYCAPTION",         "Caption bar height",
SM_CXBORDER,          "SM_CXBORDER",          "Window border width",
SM_CYBORDER,          "SM_CYBORDER",          "Window border height",
SM_CXDLGFRAME,        "SM_CXDLGFRAME",        "Dialog window frame width",
SM_CYDLGFRAME,        "SM_CYDLGFRAME",        "Dialog window frame height",
SM_CYVTHUMB,          "SM_CYVTHUMB",          "Vertical scroll thumb height",
SM_CXHTHUMB,          "SM_CXHTHUMB",          "Horizontal scroll thumb width",
SM_CXICON,            "SM_CXICON",            "Icon width",
SM_CYICON,            "SM_CYICON",            "Icon height",
SM_CXCURSOR,          "SM_CXCURSOR",          "Cursor width",
SM_CYCURSOR,          "SM_CYCURSOR",          "Cursor height",
SM_CYMENU,            "SM_CYMENU",            "Menu bar height",
SM_CXFULLSCREEN,      "SM_CXFULLSCREEN",      "Full screen client area width",
SM_CYFULLSCREEN,      "SM_CYFULLSCREEN",      "Full screen client area height",
SM_CYKANJIWINDOW,     "SM_CYKANJIWINDOW",     "Kanji window height",
SM_MOUSEPRESENT,      "SM_MOUSEPRESENT",      "Mouse present flag",
SM_CYVSCROLL,         "SM_CYVSCROLL",         "Vertical scroll arrow height",
SM_CXHSCROLL,         "SM_CXHSCROLL",         "Horizontal scroll arrow width",
SM_DEBUG,             "SM_DEBUG",             "Debug version flag",
SM_SWAPBUTTON,        "SM_SWAPBUTTON",        "Mouse buttons swapped flag",
SM_RESERVED1,         "SM_RESERVED1",         "Reserved",
SM_RESERVED2,         "SM_RESERVED2",         "Reserved",
SM_RESERVED3,         "SM_RESERVED3",         "Reserved",
SM_RESERVED4,         "SM_RESERVED4",         "Reserved",
SM_CXMIN,             "SM_CXMIN",             "Minimum window width",
SM_CYMIN,             "SM_CYMIN",             "Minimum window height",
SM_CXSIZE,            "SM_CXSIZE",            "Minimize/Maximize icon width",
SM_CYSIZE,            "SM_CYSIZE",            "Minimize/Maximize icon height",
SM_CXFRAME,           "SM_CXFRAME",           "Window frame width",
SM_CYFRAME,           "SM_CYFRAME",           "Window frame height",
SM_CXMINTRACK,        "SM_CXMINTRACK",        "Minimum window tracking width",
```

```
SM_CYMINTRACK,          "SM_CYMINTRACK",          "Minimum window tracking height",
SM_CXDOUBLECLK,         "SM_CXDOUBLECLK",         "Double click x tolerance",
SM_CYDOUBLECLK,         "SM_CYDOUBLECLK",         "Double click y tolerance",
SM_CXICONSPACING,       "SM_CXICONSPACING",       "Horizontal icon spacing",
SM_CYICONSPACING,       "SM_CYICONSPACING",       "Vertical icon spacing",
SM_MENUDROPALIGNMENT,   "SM_MENUDROPALIGNMENT",   "Left or right menu drop",
SM_PENWINDOWS,          "SM_PENWINDOWS",          "Pen extensions installed",
SM_DBCSENABLED,         "SM_DBCSENABLED",         "Double-Byte Char Set enabled",
SM_CMOUSEBUTTONS,       "SM_CMOUSEBUTTONS",       "Number of mouse buttons",
SM_SHOWSOUNDS,          "SM_SHOWSOUNDS",          "Present sounds visually"
} ;
```

图 3-4 SYSMETS.H

显示信息的程序命名为 SYSMETS1 。创建 SYSMETS1.EXE 所需要的文件(制作文件和 C 源代码)如图 3-4 中所示，现在大多数代码看起来都很熟悉。除了程序名外，制作文件和 HELLOWIN 的制作文件相同。 SYSMETS1.C 中的 WinMain 实际上与 HELLOWIN 中的 WinMain 一样。

SYSMETS1.MAK

```
#----------------------
# SYSMETS1.MAK make file
#----------------------

sysmets1.exe : sysmets1.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets1.exe sysmets1.obj $(GUILIBS)

sysmets1.obj : sysmets1.c sysmets.h
     $(CC) $(CFLAGS) sysmets1.c
```

SYSMETS1.C

```
/*----------------------------------------------------
   SYSMETS1.C -- System Metrics Display Program No. 1
                 (c) Charles Petzold, 1996
  ----------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;
     wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Get System Metrics No. 1",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;

               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;

               ReleaseDC (hwnd, hdc) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
```

```
                for (i = 0 ; i < NUMLINES ; i++)
                     {
                     TextOut (hdc, cxChar, cyChar * (1 + i),
                              sysmetrics[i].szLabel,
                              strlen (sysmetrics[i].szLabel)) ;

                     TextOut (hdc, cxChar + 22 * cxCaps, cyChar * (1 + i),
                              sysmetrics[i].szDesc,
                              strlen (sysmetrics[i].szDesc)) ;

                     SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

                     TextOut (hdc, cxChar + 22 * cxCaps + 40 * cxChar,
                              cyChar * (1 + i), szBuffer,
                              wsprintf (szBuffer, "%5d",
                              GetSystemMetrics (sysmetrics[i].iIndex))) ;
                     SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                     }
                EndPaint (hwnd, &ps) ;
                return 0 ;

           case WM_DESTROY :
                PostQuitMessage (0) ;
                return 0 ;
           }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
      }
```

图 3-5　SYSMETS1程序

图 3-6 显示了在 VGA 上运行的 SYSMETS1。在程序窗口中可以看到，屏幕宽度为 640 个象素点，屏幕高度为 480 个象素点，这两个值以及程序所显示的其它值可能会因视频显示器类型的不同而不同。

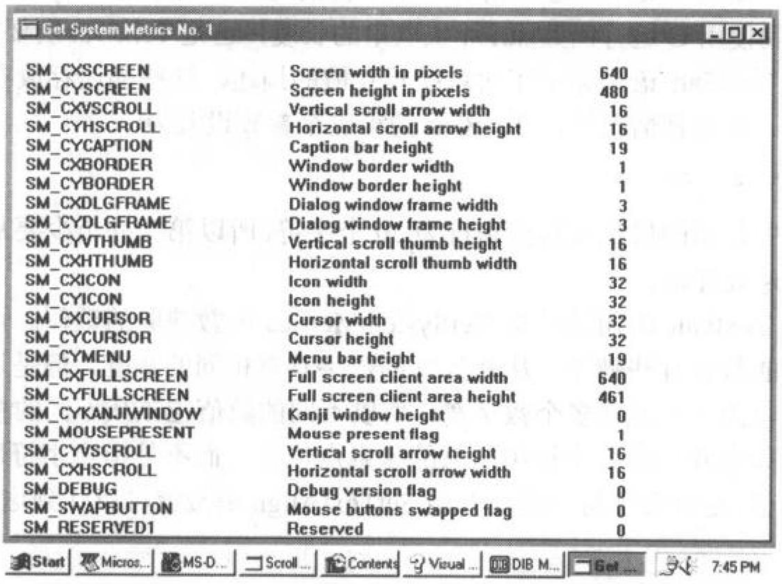


图 3-6　SYSMETS1的显示

3.2.11 SYSMETS1.C 窗口过程

SYSMETS1.C 程序中的 WndProc 窗口过程处理三条消息:WM_CREATE，WM_PAINT 和 WM_DESTROY。WM_DESTROY 消息的处理方法与第二章的 HELLOWIN 程序相同。

WM_CREATE 消息是窗口过程接收到的第一条消息。在 CreateWindow 函数创建窗口时，Windows 产生这条消息。在处理 WM_CREATE 消息时，SYSMETS1 调用 GetDC 获取窗口的设备环境，并调用 GetTextMetrics 获取缺省系统字体的文本尺寸。SYSMETS1 将平均字符宽度保存在 cxChar 中，将字符的总高度(包括外部间距)保存在 cyChar 中。

SYSMETS1 还将大写字母的平均宽度保存在静态变量 cxCaps 中。对于固定宽度的字体，cxCaps 等于 cxChar。对于可变宽度字体，cxCaps 设置为 cxChar 乘以 150%。对于可变宽度字体，TEXTMETRIC 结构中的 tmPitchAndFamily 域的低位为 1，对于固定宽度字体，该值为 0。SYSMETS1 使用这个位来计算 cxCaps:

```
cxcaps = (tm.tmPitchAndFamily & 1?3:2)*cxChar/2;
```

SYSMETS1 在处理 WM_PAINT 消息期间完成所有窗口创建工作。通常，窗口过程先调用 BeginPaint 获取设备环境句柄,然后用一条 for 语句对 SYSMETS.H 中定义的 sysmetrics 结构的所有行进行循环。三列文本用三个 TextOut 函数显示，对于每一列，TextOut 的第三个参数都设置为:

```
cyChar * (1+i)
```

这个参数指示了字符串顶部相对于客户区顶部的象素点位置。因此，程序在顶部留出 cyChar 个象素点的边空。第一行文本(当 i=0 时)从客户区顶部以下 cyChar 象素点处开始。

第一条 TextOut 语句在第一列显示了大写标识符。TextOut 的第二个参数为 cxChar。这说明客户区左边与文本串之间留有一个字符的边空。文本是从 sysmetrics 结构的 szLabel 域获取的。我使用 C 运行函数 strlen 获取串的长度，这是 TextOut 所要求的最后一个参数。

第二条 TextOut 语句显示了对系统尺寸值的描述。这些描述存放在 sysmetrics 结构的 szDesc 域中。在这种情况下，TextOut 的第二个参数设置为:

```
cxChar + 22 * cxCaps
```

第一列中显示的最长大写标识符为 20 个字符,所以第二列至少要从距离第一列文本开头 20*cxCaps 处开始。

第三条 TextOut 语句显示从 GetSystemMetrics 函数获取的数值。可变宽度字体使得格式化右对齐的数值有些棘手。从 0 到 9 的数字具有相同的宽度，但是这个宽度比空格宽度大。数值可以为一个或者多个数字宽，所以不同的数值应该从不同的横向位置开始。

那么，如果我们指定字符串结束的象素点位置，而不是指定字符串的开始位置，以此来右对齐数值,是否会容易一些呢？用 SetTextAlign 函数就可以作到这一点。在 SYSMETS1 调用:

```
SetTextAlign(hdc, TA_RIGHT | TA_TOP) ;
```

之后，传给后续 TextOut 函数的坐标将指定文本串的右上角，而不是左上角。

显示列数的 TextOut 函数的第二个参数设置为：

cxChar + 23 * cxCaps + 40 * cxChar

值 40*cxChar 包含了第二列的宽度和第三列的宽度。在 TextOut 函数之后，另一个对 SetTextAlign 的调用将对齐方式设置回普通方式，以进行下次循环。

3.2.12 空间不够

在 SYSMETS1 程序中存在着一个很难处理的问题：除非你具有一个大屏幕、高分辨率的视频适配器，否则就无法看到系统尺度列表的最后几行。如果窗口太窄，甚至根本看不到值。

SYSMETS1 不知道其客户区有多大，它从窗口顶部开始输出文本，并依靠 Windows 来裁剪客户区边界之外的内容，这很不理想。为了解决这个问题，我们的第一个任务是确定程序在客户区内能输出多少内容。

3.2.13 客户区的大小

如果读者使用过现有的 Windows 应用程序，可能会发现窗口的尺寸变化极大。窗口最大时(假定窗口没有菜单和滚动条)，客户区占据除了标题条之外的整个屏幕。这一最大化了的客户区的尺寸可以通过以 SM_CXFULLSCREEN 和 SM_CYFULLSCREEN 为参数调用 GetSystemMetrics 来获得。窗口的最小尺寸可以很小，有时甚至不存在，更不用说客户区了。

确定客户区大小的普通方法是在窗口过程内处理 WM_SIZE 消息。每当窗口的尺寸改变时，Windows 给窗口过程发送 WM_SIZE 的消息，传给窗口过程的 lParam 参数的低位字中包含客户区的宽度，高位字中包含客户区的高度。处理这一个消息的代码形式如下：

```
static int cxClient, cyClient;
[其它程序行]
case WM_SIZE:
cxClient = LOWORD(lParam) ;
cyClient = HIWORD(lParam) ;
return 0;
```

LOWORD 和 HIWORD 宏在 WINDOWS.H 中定义。与 cxChar 和 cyChar 相似， cxClient 和 cyClient 变量在窗口过程内定义为静态变量，因为在以后处理其它消息时会用到它们。

WM_SIZE 消息必然跟着一条 WM_PAINT 消息，为什么呢？因为在我们定义窗口类时指定类风格为：

```
CS_HREDRAW|CS_VREDRAW
```

这种类风格告诉 Windows，如果水平或者垂直大小发生改变，则强行刷新客户区。

用如下公式计算可以在客户区内显示的文本的总行数：

```
cyClient / cyChar
```

如果客户区的高度太小以至无法显示一个完整的字符，这个公式的结果可以为 0。类似地，

在客户区的水平方向可以显示的小写字符的近似数目为:

```
cxClient / cxChar
```

如果在处理 WM_CREATE 消息期间确定 cxChar 和 cyChar，则不用担心在这两个计算公式中会出现被 0 除的情况。在 WinMain 调用 CreateWindow 时，窗口过程接收一条 WM_CREATE 消息。在 WinMain 调用 ShowWindow 之后接收到第一条 WM_CREATE 消息，此时 cxChar 和 cyChar 已经被赋予正的非零值了。

如果客户区的大小不足以容纳所有的内容，那么，知道窗口客户区的大小只是为用户提供了在客户区内移动文本的第一步。如果读者对其它有类似需求的 Windows 应用程序很熟悉，就很可能知道，这种情况下，我们需要使用"滚动条"。

3.3 滚动条

滚动条是图形和鼠标界面中最好的功能之一，它很容易使用，而且提供了很好的视觉反馈。显示任何东西——无论是文本、图形、表格、数据库记录还是图象时，只要它所需的空间超出了窗口的客户区所能提供的空间，就可以使用滚动条。

滚动条既有垂直方向的（供上下移动），也有水平方向的(供左右移动)。用户可以使用鼠标在滚动条两端的箭头上或者在箭头之间的区域中点一下，这时，"滚动框"在滚动条内的移动模拟了所显示的信息在整个文档中的近似相关位置。用户也可以使用鼠标拖动滚动框移动到一个特定的位置。图 3-7 显示了推荐使用的垂直滚动条。

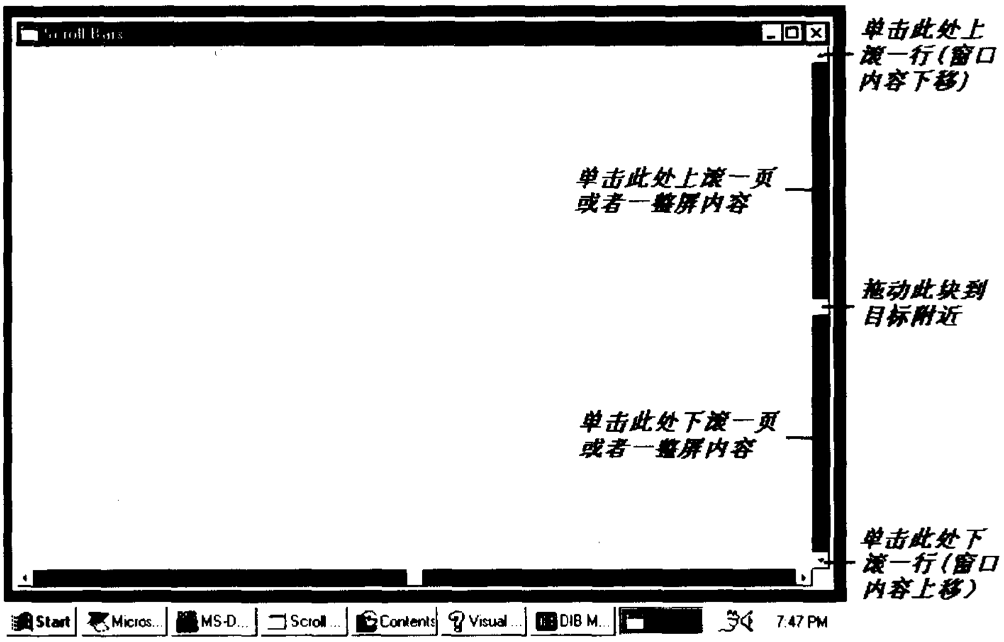


图 3-7 垂直滚动条

有时，程序员对滚动概念很难理解，因为他们的观点与用户的观点不同：用户向下滚动是想看到文档稍下的部分；但是，程序实际上是将文档相对于显示窗口向上移动。

Windows 文档和 WINDOWS.H 标识符是基于用户的观点：向上滚动意味着朝文档的开头移动；向下滚动意味着朝文档尾部移动。

很容易在应用程序中包含水平或者垂直的滚动条，程序员只需要在 CreateWindow 语句的窗口风格中包含标识符 WS_VSCROLL(垂直滚动)和/或 WS_HSCROL(水平滚动)即可。这些滚动条通常放在窗口的右部和底部，伸展为客户区的整个长度或宽度。客户区不包含滚动条所占据的空间。对于特定的显示驱动程序，垂直滚动条的宽度和水平滚动条的高度是恒定的。如果需要这些值，可以使用 GetSystemMetrics 调用来获取（如前面的程序那样）。

Windows 负责处理对滚动条的所有鼠标操作，但是，窗口滚动条没有自动的键盘界面。如果想用光标键来完成滚动功能，则必须提供这方面的代码（我们将在第 5 章讨论键盘时做到这一点）。

3.3.1 滚动条的范围和位置

每个滚动条均有一个相关的“范围”(这是一对整数，分别代表最小值和最大值)和“位置”(它是滚动框在此范围内的位置)。当滚动框在滚动条的顶部（或左部)时，滚动框的位置是范围的最小值；在滚动条的底部（或右部)时，滚动框的位置是范围的最大值。

在缺省情况下，滚动条的范围是从 0(顶部或左部）至 100(底部或右部），但将范围改变为更方便于程序的数值也是很容易的：

```
SetScrollRange(hwnd, iBar, iMin, iMax, bRedraw) ;
```

参数 iBar 为 SB_VERT 或者 SB_HORZ， iMin 和 iMax 分别是范围的最小值和最大值。如果想要 Windows 根据新范围重画滚动条，则设置 bRedraw 为 TRUE 。

滚动框的位置总是离散的整数值。例如，范围为 0 至 4 的滚动条具有五个滚动框位置，如图 3-8 所示。你可以使用 SetScrollPos 在滚动条范围内设置一个新的滚动框位置：

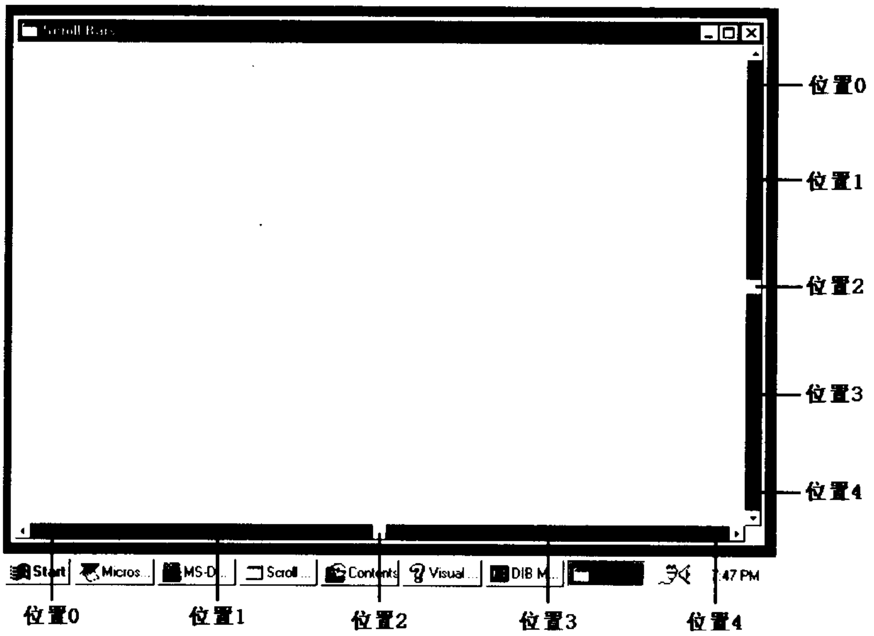


图 3-8 具有5个滚动框位置的滚动条

```
SetScrollPos(hwnd, iBar, iPos, bRedraw) ;
```

参数 iPos 是新位置，它必须在 iMin 至 iMax 的范围内。 Windows 提供了类似的函数(GetScrollRange 和 GetScrollPos)来获取滚动条的当前范围和位置。

在程序内使用滚动条时，程序员与 Windows 共同负责维护滚动条以及更新滚动框的位置。下面是 Windows 对滚动条的处理：

- 处理所有滚动条鼠标事件
- 当用户在滚动条内点一下时，提供一种“反相视频”闪烁
- 当用户在滚动条内拖动滚动框时，显示一个“阴影”框
- 为包含滚动条窗口的窗口过程发送滚动条消息

以下是程序员应该完成的工作：

- 初始化滚动条的范围
- 处理滚动条消息
- 更新滚动条内滚动框的位置

3.3.2 滚动条消息

在用鼠标点一下滚动条或者拖动滚动框时， Windows 给窗口过程发送 WM_VSCROLL 和 WM_HSCROLL 消息。在滚动条上的每个鼠标动作都至少产生两条消息，一条在按下鼠标按钮时产生，一条在释放按钮时产生。

WM_VSCROLL 和 WM_HSCROLL 消息中参数 wParam 的低位字是一个数值，它指示鼠标对滚动条所做的动作。 wParam 中的这些值对应 WINDOWS.H 标识符，以 SB_打头， SB 代表“ scroll bar ” (滚动条)。尽管其中有些标识符使用“ UP ” 或者“ DOWN ”，但它们同时适用于垂直滚动条和水平滚动条，如图 3-9 所示。如果在滚动条内按下鼠标按钮，窗口过程能接收多个 SB_LINEUP 、 SB_PAGEUP 、 SB_LINDOWN 或者 SB_PAGEDOWN 消息。SB_ENDSCROLL 消息标志着鼠标已经被释放，一般可以忽略 SB_ENDSCROLL 消息。

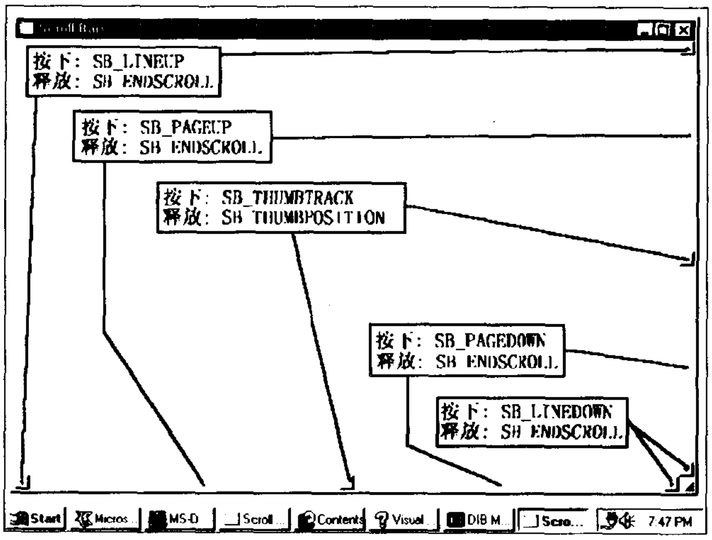


图 3-9 用于滚动条消息的wParam值的标识符

如果 wParam 的低位字是 SB_THUMBTRACK 或者是 SB_THUMBPOSITION ，则 wParam 的高位字是拖动滚动条的当前位置，这个位置在滚动条范围的最小值与最大值之间。对于其它滚动条动作， wParam 的高位字应该被忽略。你也可以忽略 lParam ；它通常用于在对话框中创建的滚动条。

Windows 文档中指出， wParam 的低位字也可以为 SB_TOP 或者 SB_BOTTOM ，这说明滚动条已经被移至最小或者最大位置。但是，对于作为应用程序窗口一部分而创建的滚动条来说，永远不会接收到这些值。

处理 SB_THUMBTRACK 和 SB_THUMBPOSITION 是困难的。如果设置的滚动条很大，用户在滚动条内很迅速地拖动滚动框，则 Windows 给窗口函数发送一连串的 SB_THUMBTRACK 消息。程序要保存这些消息可能有困难。鉴于此，大多数 Windows 应用程序忽略这些消息，仅在接收到 SB_THUMBPOSITION 时采取行动， SB_THUMBPOSITION 消息意味着滚动框又静止了。

尽管如此，如果程序能够快速地刷新显示，就可以在程序中包含对 SB_THUMBTRACK 的处理。但是，要知道，当发现程序能随滚动框的移动而迅速上下滚动时，用户毫无疑问会尽可能快地移动滚动框，这时，如果滚动条反应太慢，用户就会不满意。

3.3.3 在 SYSTEMS 中加入滚动功能

前面的说明已经很详尽了，现在，要将其应用于实践中。让我们开始时简单些，从垂直滚动入手，因为我们实在太需要垂直滚动了，而暂时还可以不用水平滚动。 SYSMET2 如图 3-10 所示。

新的 CreateWindow 调用在 CreateWindow 调用中包含 WS_VSCROLL 窗口风格，从而在窗口中加入了垂直滚动，其窗口风格为:

```
WS_OVERLAPPEDWINDOW|WS_VSCROLL
```

SYSMETS2.MAK

```
#-----------------------
# SYSMETS2.MAK make file
#-----------------------

sysmets2.exe : sysmets2.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets2.exe sysmets2.obj $(GUILIBS)

sysmets2.obj : sysmets2.c sysmets.h
     $(CC) $(CFLAGS) sysmets2.c
```

SYSMETS2.C

```
/*-----------------------------------------------------
   SYSMETS2.C -- System Metrics Display Program No. 2
                 (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
```

```
#include <string.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Get System Metrics No. 2",
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar, cyClient, iVscrollPos ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i, y ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
```

```
{
case WM_CREATE :
     hdc = GetDC (hwnd) ;

     GetTextMetrics (hdc, &tm) ;
     cxChar = tm.tmAveCharWidth ;
     cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     ReleaseDC (hwnd, hdc) ;

     SetScrollRange (hwnd, SB_VERT, 0, NUMLINES, FALSE) ;
     SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;
     return 0 ;
case WM_SIZE :
     cyClient = HIWORD (lParam) ;
     return 0 ;

case WM_VSCROLL :
     switch (LOWORD (wParam))
          {
          case SB_LINEUP :
               iVscrollPos -= 1 ;
               break ;

          case SB_LINEDOWN :
               iVscrollPos += 1 ;
               break ;

          case SB_PAGEUP :
               iVscrollPos -= cyClient / cyChar ;
               break ;

          case SB_PAGEDOWN :
               iVscrollPos += cyClient / cyChar ;
               break ;

          case SB_THUMBPOSITION :
               iVscrollPos = HIWORD (wParam) ;
               break ;

          default :
               break ;
          }
     iVscrollPos = max (0, min (iVscrollPos, NUMLINES)) ;

     if (iVscrollPos != GetScrollPos (hwnd, SB_VERT))
          {
          SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          }
     return 0 ;

case WM_PAINT :
     hdc = BeginPaint (hwnd, &ps) ;
```

```
          for (i = 0 ; i < NUMLINES ; i++)
               {
               y = cyChar * (1 - iVscrollPos + i) ;

               TextOut (hdc, cxChar, y,
                        sysmetrics[i].szLabel,
                        strlen (sysmetrics[i].szLabel)) ;

               TextOut (hdc, cxChar + 22 * cxCaps, y,
                        sysmetrics[i].szDesc,
                        strlen (sysmetrics[i].szDesc)) ;

               SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

               TextOut (hdc, cxChar + 22 * cxCaps + 40 * cxChar, y,
                        szBuffer,
                        wsprintf (szBuffer, "%5d",
                        GetSystemMetrics (sysmetrics[i].iIndex))) ;

               SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
               }

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 3-10 SYSMET2程序

WndProc 窗口过程在处理 WM_CREATE 消息时增加了两条语句，以设置垂直滚动条的范围和位置:

```
SetScrollRange(hwnd, SB_VERT, 0, NUMLINES, FALSE) ;
SetScrollPos(hwnd, SB_VERT, iVscrollPos, TRUE) ;
```

sysmetrics 结构具有 NUMLINES 行文本，所以滚动条范围被设置为 0 至 NUMLINES 。滚动条的每个位置对应于在客户区顶部显示的一个文本行。如果滚动框的位置为 0，则在屏幕的顶部留出一个空行作为边距。在向下滚动增大滚动条的位置时，文本必须向上移动。当滚动条的位置位于底部时，结构的最后一行位于顶部。

为了有助于处理 WM_SCROLL 消息，在 WinProc 窗口过程中定义了一个静态变量 iVscrollPos ，这一变量是滚动条内滚动框的当前位置。对于 SB_LINEUP 和 SB_LINEDOWN ，只需要将滚动框调整一个单位的位置。对于 SB_PAGEUP 和 SB_PAGEDOWN ，我们想整屏地移动屏幕的内容，或者移动 cyClient/cyChar 个单位的位置。对于 SB_THUMBPOSITION ，新的滚动框位置是 wParam 的高位字。 SB_ENDSCROLL

和 SB_THUMBTRACK 消息被忽略。

然后，用 min 和 max 宏来调整 iVscrollPos，以确保它在最大值与最小值之间。如果滚动位置发生了变化，则使用 SetScrollPos 来进行更新，并且调用 InvalidateRect 使整个窗口无效。

InvalidateRect 调用产生一条 WM_PAINT 消息。SYSMETS1 在处理 WM_PAINT 消息时，每一行的 y 坐标计算公式为:

```
cyChar * (1+i)
```

在 SYSMETS2 中，计算公式为:

```
cyChar * (1-nVscrollPos+i)
```

循环仍然显示 NUMLINES 行文本，但是对于 nVscrollPos 的值为 2 或者更大时，循环开始在客户区之上显示文本。由于越过了客户区，Windows 不显示这些行。

前面说过，我们一开始不想弄得太复杂，这样的代码很浪费，效率很低。下面我们对此加以修改，但是先要考虑在 WM_VSCROLL 消息之后如何刷新客户区。

3.3.4 绘图程序的组织

在处理完滚动条消息后，SYSMETS2 不刷新客户区，相反，它调用 InvalidRect 使客户区失效。这导致 Windows 将一条 WM_PAINT 消息放入消息队列中。

最好能使 Windows 程序在响应 WM_PAINT 消息时完成所有的客户区绘制功能。因为程序必须在一接收到 WM_PAINT 消息时就刷新整个客户区，如果在程序的其它部分也绘制的话，将很可能使代码重复。

你或许不能一下子接受这样的做法，因为这与普通 PC 程序设计非常不同。不可否认，在响应非 WM_PAINT 消息时绘制客户区偶尔也会很方便（第五章中的 KEYLOCK 程序就是一例），但是在很多情况下，这完全是不必要的。当你掌握了在响应 WM_PAINT 消息时积累绘制客户区所需要的全部信息这样的原则之后，会对这种结果感到满意的。不过，当你的程序仍然需要在处理非 WM_PAINT 消息时刷新特定的显示区域，使用 InvalidateRect 就很方便，你可以用它使客户区内的特定矩形或者整个客户区失效。

只将窗口客户区标记为无效以产生 WM_PAINT 消息，对于某些应用程序来说也许不是完全令人满意的选择。在调用 InvalidateRect 之后，Windows 将 WM_PAINT 消息放入消息队列中，最后由窗口过程处理它。然而，Windows 将 WM_PAINT 消息当成低优先级消息，如果系统有许多其它的动作正在发生，那么也许会让你等待一会儿工夫。这时，当对话框消失时，将会出现一些空白的“洞”。

如果你希望立即刷新失效区域，可以在调用 InvalidateRect 之后调用 UpdateWindow:

```
UpdateWindow(hwnd) ;
```

如果客户区的任一部分无效，则 UpdateWindow 将导致 Windows 用 WM_PAINT 消息调用

窗口过程(如果整个客户区有效，则不调用窗口过程)。这一 WM_PAINT 消息不进入消息队列，直接由 Windows 调用窗口过程。窗口过程完成刷新后立即退出， Windows 将控制返回给程序中 UpdateWindow 调用之后的语句。

读者可能注意到， UpdateWindow 与 WinMain 中用来产生第一条 WM_PAINT 消息的函数相同。最初创建窗口时，整个客户区变为失效， UpdateWindow 指示窗口过程绘制客户区。

3.3.5 建立更好的滚动条

由于 SYSMETS2 效率太低，限制了它在其他程序中的使用，所以我们提供 SYSMETS 程序在这一章中的最后一个版本 SYSMETS3 ，如图 3-11 所示。该版本增加了一个水平滚动条，供左右滚动文本，刷新客户区的效率也提高了。

SYSMETS3.MAK

```
#-----------------------
# SYSMETS3.MAK make file
#-----------------------

sysmets3.exe : sysmets3.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets3.exe sysmets3.obj $(GUILIBS)

sysmets3.obj : sysmets3.c sysmets.h
     $(CC) $(CFLAGS) sysmets3.c
```

SYSMETS3.C

```
/*----------------------------------------------------
   SYSMETS3.C -- System Metrics Display Program No. 3
                 (c) Charles Petzold, 1996
  ----------------------------------------------------*/

#include <windows.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets3" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
```

```
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Get System Metrics No. 3",
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth,
                 iVscrollPos, iVscrollMax, iHscrollPos, iHscrollMax ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i, x, y, iPaintBeg, iPaintEnd, iVscrollInc, iHscrollInc ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;

               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;

               ReleaseDC (hwnd, hdc) ;

               iMaxWidth = 40 * cxChar + 22 * cxCaps ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
```

```
          iVscrollMax = max (0, NUMLINES + 2 - cyClient / cyChar) ;
          iVscrollPos = min (iVscrollPos, iVscrollMax) ;

          SetScrollRange (hwnd, SB_VERT, 0, iVscrollMax, FALSE) ;
          SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;

          iHscrollMax = max (0, 2 + (iMaxWidth - cxClient) / cxChar) ;
          iHscrollPos = min (iHscrollPos, iHscrollMax) ;

          SetScrollRange (hwnd, SB_HORZ, 0, iHscrollMax, FALSE) ;
          SetScrollPos   (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
          return 0 ;

     case WM_VSCROLL :
          switch (LOWORD (wParam))
               {
               case SB_TOP :
                    iVscrollInc = -iVscrollPos ;
                    break ;

               case SB_BOTTOM :
                    iVscrollInc = iVscrollMax - iVscrollPos ;
                    break ;

               case SB_LINEUP :
                    iVscrollInc = -1 ;
                    break ;

               case SB_LINEDOWN :
                    iVscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    iVscrollInc = min (-1, -cyClient / cyChar) ;
                    break ;

               case SB_PAGEDOWN :
                    iVscrollInc = max (1, cyClient / cyChar) ;
                    break ;

               case SB_THUMBTRACK :
                    iVscrollInc = HIWORD (wParam) - iVscrollPos ;
                    break ;

               default :
                    iVscrollInc = 0 ;
               }
          iVscrollInc = max (-iVscrollPos,
                        min (iVscrollInc, iVscrollMax - iVscrollPos)) ;

          if (iVscrollInc != 0)
               {
               iVscrollPos += iVscrollInc ;
               ScrollWindow (hwnd, 0, -cyChar * iVscrollInc, NULL, NULL) ;
```

```
               SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
               UpdateWindow (hwnd) ;
               }
          return 0 ;

     case WM_HSCROLL :
          switch (LOWORD (wParam))
               {
               case SB_LINEUP :
                    iHscrollInc = -1 ;
                    break ;

               case SB_LINEDOWN :
                    iHscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    iHscrollInc = -8 ;
                    break ;

               case SB_PAGEDOWN :
                    iHscrollInc = 8 ;
                    break ;

               case SB_THUMBPOSITION :
                    iHscrollInc = HIWORD (wParam) - iHscrollPos ;
                    break ;

               default :
                    iHscrollInc = 0 ;
               }
          iHscrollInc = max (-iHscrollPos,
                        min (iHscrollInc, iHscrollMax - iHscrollPos)) ;

          if (iHscrollInc != 0)
               {
               iHscrollPos += iHscrollInc ;
               ScrollWindow (hwnd, -cxChar * iHscrollInc, 0, NULL, NULL) ;
               SetScrollPos (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
               }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          iPaintBeg = max (0, iVscrollPos + ps.rcPaint.top / cyChar - 1) ;
          iPaintEnd = min (NUMLINES,
                           iVscrollPos + ps.rcPaint.bottom / cyChar) ;

          for (i = iPaintBeg ; i < iPaintEnd ; i++)
               {
               x = cxChar * (1 - iHscrollPos) ;
               y = cyChar * (1 - iVscrollPos + i) ;

               TextOut (hdc, x, y,
```

```
                        sysmetrics[i].szLabel,
                        strlen (sysmetrics[i].szLabel)) ;

                TextOut (hdc, x + 22 * cxCaps, y,
                        sysmetrics[i].szDesc,
                        strlen (sysmetrics[i].szDesc)) ;

                SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y,
                        szBuffer,
                        wsprintf (szBuffer, "%5d",
                            GetSystemMetrics (sysmetrics[i].iIndex))) ;

                SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                }

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }

    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 3-11 SYSMETS3程序

下面列出了 SYSMETS3 的改进之处以及在程序中的实现细节:

- 你不再能够滚动到最后一行显示在客户区的顶部，只能滚动到足够在客户区底部看到最后一行文本为止。这就要求程序在处理 WM_SIZE 消息时计算新的滚动条范围(以及新的滚动框位置)。在 WM_SIZE 的处理中，根据文本行数、宽度和客户区的大小来计算滚动条范围。这种方法会得到更小的范围——只有显示客户区中文本所必需的大小。

  这种做法有一个意外收获。假设窗口的客户区足够大，能在容纳顶边空和底边空的情况下显示所有文本，在这种情况下，滚动条范围的最小位置和最大位置都等于 0 。 Windows 将对这个信息做些什么呢？它将从窗口内擦除滚动条！因为不再需要滚动条了。类似地，如果客户区足够宽，可以显示整个 60 列文本，在窗口中就不再显示水平滚动条。

- WM_VSCROLL 和 WM_HSCROLL 消息的处理是先对每个 wParam 值计算滚动条位置的增量，然后将这个值用在 ScrollWindow 调用中，以滚动窗口内已经存在的内容。函数的格式为:

  ```
  ScrollWindow(hwnd, xInc, yInc, lpRect, lpClipRect) ;
  ```

  xInc 和 yInc 的值指定了要滚动的象素点数。在 SYSMETS3 中，设置 lpRect 和 lpClipRect 的值为 NULL ，以指定必须滚动整个客户区。 Windows 使客户区中因滚动操作而“暴露”的矩形失效。不再需要 InvalidateRect (注意， ScrollWindow

不是 GDI 过程，因为它不需要设备环境句柄。它是能够改变窗口客户区外观的 Windows 函数中少数几个非 GDI 函数之一)。

- 对 WM_PAINT 的处理则确定哪些行在无效矩形中，并且只重写这些行。它通过分析 PAINTSTRUCT 结构中保存的无效矩形的顶部和底部坐标来作到这一点。程序仅绘制无效矩形内的文本行，代码稍微复杂一些，但是速度要快得多。
- 由于 WM_PAINT 速度加快了，我们决定让 SYSMETS3 处理 WM_VSCROLL 消息的 SB_THUMBTRACK 操作。前面，程序将会忽略 SB_THUMBTRACK 消息(它在用户拖动滚动框时产生)，只响应 SB_THUMBPOSITION 消息，SB_THUMBPOSITION 是在用户停止拖动滚动框后产生的。WM_VSCROLL 代码也调用 UpdateWindow，以便立即刷新客户区。在垂直滚动条内移动滚动框时，SYSMETS3 将连续地滚动和刷新客户区。我们相信读者会判断出 SYSMETS3 的速度是否足够的快。

3.3.6 不用鼠标怎么办

在 Windows 的早期，有大量的用户不喜欢使用鼠标，而且，Windows 自身也不要求必须有鼠标。虽然，没有鼠标的 PC 现在走上了单色显示器和点阵打印机的没落之路，但我仍然建议你编写可以使用键盘来模拟鼠标操作的程序，尤其对于像滚动条这样的基本操作对象更是如此。因为我们的键盘有一组光标移动键，所以应该实现同样的操作。

在第 5 章中，你将学习如何使用键盘以及如何在 SYSMET3 中增加键盘界面。你可能会注意到，SYSMETS3 似乎处理了 WM_VSCROLL 消息中 wParam 的低位字等于 SB_TOP 和 SB_BOTTOM 的情形。前面已经提到过，窗口过程不从滚动条接收这些消息，所以，目前这是多余的代码。当我们在第 5 章再次回到这个程序时，你将会明白这样做的原因。

第4章 图形基础

图形设备界面（GDI）是 Windows 95 的子系统，它负责在视频显示器和打印机上显示图形（包括文本）。正如你可能想象的那样，GDI 是 Windows 非常重要的部分。不但你为 Windows 95 编写的应用系统在显示视觉信息时只使用 GDI，而且，就是 Windows 本身也只使用 GDI 来显示用户界面对象，诸如菜单、滚动条、图标和设备光标。

可能由于“怎么干都行”的 MS_DOS 编程传统，因此经常有一些 Windows 编程新手（甚至有时还会有一些老手）会不自觉地“溜过” GDI 而直接地将输出写到视频显示器。请千万别这样做，甚至别这样想！它只会给你带来许多的麻烦，包括对其它 Windows 95 程序的干扰，以及和将来的操作系统不兼容等问题。

与 Windows 95 的某些新特征不同，GDI 已经有了非常长的历史。Windows 1.0 内部基本上包含了三个动态链接库：KERNEL（处理任务调度、存储管理和文件 I/O）、USER（用户界面）和 GDI。这以后的 Windows 在 GDI 中增加了一些功能，同时（在大多数情况下）保持和现有程序的向后兼容，但是 GDI 的内核却没有变。

不幸的是，如果要对 GDI 进行全面的讲述将要求一整本书——当然不是这本书。在本章中，我只是想给你提供足够的 GDI 知识，以理解本书后面演示的绘图例程，同时为你提供一些可能对你有用的其它信息。

4.1 GDI 原 理

在 32 位 Windows 中的图形主要由 GDI32.DLL 动态链接库输出的函数来处理，这些函数相应地使用 16 位的动态链接库 GDI.EXE（早期 Windows 版本中的动态链接库的扩展名为.EXE 而不是.DLL）。这些函数调用各种显示驱动文件（一个视频显示屏幕的.DRV 文件，以及一个或者多个控制打印机或者绘图仪的其它.DRV 文件）。视频驱动访问视频显示器的硬件，或者将 GDI 命令转换为各种打印机能够理解的代码或者命令。不同的视频显示适配器和打印机要求不同的驱动文件。

因为 PC 兼容机上可以连接许多种不同的视频设备，所以，GDI 的主要目的之一是支持设备无关的图形。Windows 程序应该能够毫无困难地在 Windows 支持的任何一种输出设备上运行，GDI 通过将你的程序和不同输出设备的特殊特性隔离开来的方法来达到这一目的。

图形输出设备被分为两大类：光栅设备和矢量设备。大多数 PC 输出设备是光栅设备，这意味着它们以点模式来表示图象，这一类包括视频显示适配器、点阵打印机和激光打印机。矢量设备使用线来绘制图象，通常局限于绘图仪。

许多传统的计算机图形编程都是完全基于矢量的，这意味着一个使用某种图形语言的

程序是在硬件之上一定层次的抽象。输出设备用象素表示图形，但是程序与编程接口之间并不是用象素进行对话的。你当然可以使用 Windows GDI 作为一个高层次的矢量绘制系统，同时也可以将它用于比较低层次的象素操作。

从这方面来看，Windows GDI 和传统的图形接口语言之间的关系，就如同 C 和其它编程语言之间的关系一样。C 以它在不同操作系统和环境之间的高度可移植性而闻名，然而 C 也以它允许一个程序员进行低层次系统调用而闻名，这些调用在其它高级语言中通常是不可能的。正如 C 有时被认为是一种“高级汇编语言”一样，你可以认为 GDI 是和图形设备硬件之间的一种高层接口。

你已经看到，Windows 缺省地使用基于象素的坐标系。大多数传统的图形语言使用“虚拟”坐标系，其水平和垂直轴的范围在 0 到 32,767 之间。虽然有些图形语言不让你使用象素坐标，但是 Windows GDI 却允许你使用两者之一（以及基于物理度量的坐标系）。你可以使用一种虚拟坐标系从而使你的程序远离硬件，或者也可以使用设备坐标系而完全和硬件接近。

某些程序员认为当你开始基于象素的程序设计之后，你就抛弃了设备无关性。我们已经在第三章中看到，这并不总是正确的，其中的诀窍是在一种与设备无关的方式中使用象素。这要求图形接口语言为程序提供一些方法来确定设备的硬件特征，并进行适当的调节。例如，在 SYSMETS 程序中，我们根据标准系统字体字符的象素大小来确定屏幕上文本的间距，这种方法允许程序针对不同分辨率、不同文本大小和不同方向比例的不同显示适配器进行相应的调节。你将在本章中看到一些其它用于确定显示尺寸的方法。

在早期，许多用户在一种单色显示器上运行 Windows；即使在近几年，笔记本计算机还只有灰度级别。为此，GDI 的设计保证了你可以在编写一个程序时不必过多地担心色彩问题——也就是说，Windows 可以将色彩转换为灰度级别。即使在今天，Windows 95 使用的视频显示有了不同的色彩能力（16 色、256 色或者“真彩色”），而大多数用户仍然坚持使用黑白打印机。盲目地使用这些设备是可以的，但是你的程序也应该能决定在某种显示设备上有多少色彩可以使用，从而最好地利用硬件。

当然，就如同你编写 C 程序时，为了使它在其它计算机上运行而遇到一些微妙的移植性问题一样，你也可能不小心让设备依赖性溜进你的 Windows 程序，这就是不与硬件完全隔离的代价。我们将在本章中检查一些这样的设备有关的陷井。

你还应该知道 Windows GDI 的局限，GDI 并不能做到你想对一个图形界面作的任何事。虽然你可以在显示器上到处移动图形对象，但 GDI 通常是一个静态的显示系统，只有有限的动画支持。在 Winows 95 中实现的 GDI 没有提供对三维表示和对象旋转的直接支持。例如，当你绘制一个椭圆时，椭圆轴必须分别平行于水平轴和垂直轴。虽然某些图形语言使用浮点数作为虚拟坐标，但是由于性能的原因，Windows 95 总是使用 16 位的有符号整数。不幸的是，这是 Windows 95 的一个缺陷。Windows NT 支持 32 位坐标。

4.2 GDI 的结构

从程序员的观点来看，GDI 由几百个函数调用和一些相关的数据类型、宏和结构组成，

但是在我们开始讲述这些函数的细节之前，让我们先从宏观上了解一下 GDI 的整体结构。

4.2.1 函数调用的类型

一般来说，与 GDI 有关的函数调用可以分为几大类，这样的划分并非十分清晰，而会有一些这样那样的重叠：

- 获取（或者创建）和释放（或者清除）设备环境的函数——我们在第三章中已经看到过，你在绘图时需要设备环境句柄。GetDC 和 RealseDC 函数让你在非 WM_PAINT 的消息期间来作到这一点，而 BeginPaint 和 EndPaint 函数（虽然在技术上它们是 Windows 中 USER 子系统的一部分）在进行绘图的 WM_PAINT 消息期间使用。我们马上还会介绍有关设备环境的其它一些函数。
- 获取有关设备环境信息的函数——再以第三章中 SYSMETS 程序为例，我们使用 GetTextMetrics 函数来获取有关设备环境中当前选择的字体的尺寸信息。在本章后面，我们将看到一个获取非常广泛的设备环境信息的 DEVCAPS1 程序。
- 绘图函数——显然，在所有前提条件都得以满足之后，这些函数是真正重要的部分。在第三章中，我们使用 TextOut 函数在窗口的客户区中显示一些文本。我们将看到，其它 GDI 函数还可以让你画线、填充区域以及创建位图图象。
- 设置和获取设备环境的函数——设备环境的“属性”决定有关绘图函数如何工作的细节。例如，你使用 SetTextColor 来指定你使用 TextOut （或者其它文本输出函数）绘制的文本的色彩。在第三章的 SYSMETS 程序中，我们使用 SetTextAlign 来告诉 GDI： TextOut 函数中的文本串的开始位置应该在串的右边而不是缺省的左边。设备环境的所有属性都有缺省值，当获取设备环境时这些缺省值被设置。对于所有的 Set 函数，都有相应的 Get 函数，允许你获取当前设备环境属性。
- 使用 GDI 对象的函数—— GDI 在这里变得有点混乱。首先举一个例子：你缺省地使用 GDI 绘制的所有直线都是实线并具有一个标准的宽度。你可能希望绘制更细的直线，或者是由一系列的点或者短划线组成的直线。这种线的宽度和这种线的风格不是设备环境的属性，而是一个“逻辑画笔”的特征。你可以通过在 CreatePen 、 CreatePenIndirect 或者 ExtCreatePen 函数中指定这些特征来创建一个逻辑画笔，这些函数返回一个逻辑画笔的句柄（虽然这些函数被认为是 GDI 的一部分，但是和大多数 GDI 函数调用不一样，它们不要求设备环境的句柄）。要使用这个画笔，你“选中”画笔句柄到设备环境中，这样，你画的任何线都使用这个画笔，然后，你可以取消设备环境中的画笔选择，并清除画笔对象。除了画笔以外， GDI 对象还创建用于封闭区域填充的刷子，创建字体、位图以及用于本章将涉及的 GDI 的其它一些方面。

4.2.2 GDI 图元

你在屏幕或者打印机上显示的图形类型本身可以被分为几类，通常被称为“图元

(primitives)”，它们是：

- 直线和曲线——线条是所有矢量图形绘制系统的基础。GDI 支持直线、矩形、椭圆（包括椭圆的子集圆）、椭圆圆周上的部分曲线即所谓“弧”以及 Bezier 曲线，我们将在本章中分别对它们进行介绍。所有更复杂的曲线可由折线(polyline)代替，折线通过一组非常短的直线定义一条曲线。线条用设备环境中选中的当前画笔绘制。
- 填充区域——当一系列直线或者曲线封闭了一个区域时，该区域可以使用当前 GDI 画刷对象进行填充。这个画刷可以是一种实心色彩、一种图案（可以是一系列的水平、垂直或者对角标记）或者是一个在区域内垂直或者水平重复的位图图象。
- 位图——位图是一个位的矩形数组，这些位相应于一个显示设备上的象素，它们是光栅图形的基础工具。位图通常用于在视频显示器或者打印机上显示复杂（一般都是真实的）图象。位图还可以用于显示必须很快绘制的小图象（诸如图标、鼠标光标以及在应用工具条中出现的按钮等）。GDI 支持两种类型的位图——老的（虽然仍然是非常有用的）“设备有关”位图，这是一个 GDI 对象，和新的（如 Windows 3.0 的）“设备无关”位图（或者 DIB），可以被存储在磁盘文件中。
- 文本——文本的数学味道不像计算机图形的其它方面那样浓，我们知道，文本和几百年的传统印刷术有关，它被许多的印刷工人看作为一门艺术。由于这个原因，文本通常不仅是所有的计算机图形系统中最复杂的部分，而且（如果识字还是社会的基本要求的话）也是最重要的部分。用于定义 GDI 字体对象和获取字体信息的数据结构是 Windows 中最庞大的部分之一。从 Windows 3.1 开始，GDI 开始支持 TrueType 字体，该字体是在填充轮廓线基础上创建的，这样的填充轮廓线可由其它 GDI 函数处理。基于兼容性和存储大小的考虑，Windows 95 继续支持旧式的基于位图的字体（例如缺省系统字体）。

4.2.3 其它方面

GDI 的其它方面无法这么容易地被分类，它们是：

- 映射模式和变换——虽然缺省地你是以象素为单位进行绘图，但是你并非局限于此。GDI 映射模式允许你以英寸（或者甚至以几分之英寸）或者毫米或者任何你想使用的单位来绘图（Windows NT 还支持传统的以 3x3 矩阵表示的“坐标变换”，这允许倾斜和旋转图形对象。不幸的是，在 Windows 95 中不支持坐标变换）。
- 元文件——元文件是以二进制形式存储的 GDI 命令的集合。元文件主要用于通过剪贴板传输矢量图形表示。
- 区域——区域是任何形状的一个复杂范畴，通常被定义为更简单区域的一种布尔组合。在 GDI 内部，区域除了存储为最初用来定义区域的线条的组合外，还以一系列扫描线的形式存储。你可以将区域用于绘制轮廓、填充和剪裁。
- 路径——路径是 GDI 内部存储的直线和曲线的集合。路径可以用于绘图、填充和

剪裁，还可以被转换为区域。

- 剪裁——绘图可以被限制在客户区的某一部分中，这就是所谓剪裁。剪裁区域可以是矩形或非矩形（定义为一系列的短线条），剪裁通常是通过区域或者路径来定义的。
- 调色板——定制调色板的使用通常限于显示256色的显示器。Windows仅保留这些色彩之中的20种由系统使用，你可以改变其它的236种色彩，以准确地显示以位图形式存储的真实图象。
- 打印——虽然本章限于视频显示，但是你在本章中所学到的全部知识都适用于打印（关于打印的讨论，请参见第15章）。

4.3 设 备 环 境

在我们开始绘图之前，让我们比第3章更精确地讨论一下设备环境。

当你想在一个图形输出设备（诸如屏幕或者打印机）上绘图时，你首先必须获得一个设备环境（或者DC）的句柄。在返回这个句柄给你的程序时，Windows也就给了你使用设备的权限。然后你在GDI函数中以这个句柄为一个参数，向Winddows标识你想在其上进行绘图的设备。

设备环境中包含许多确定GDI函数如何在设备上工作的当前“属性”，这些属性允许传递给GDI函数的参数只包含起始坐标或者尺寸信息，而不必包含Windows在设备上显示对象时需要的所有其它信息。例如，当你调用TextOut时，你只需要在函数中给出设备环境句柄、起始坐标、文本和文本的长度。你不必指定字体、文本色、文本后面的背景色彩，以及字符间距，因为这些属性都是设备环境的一部分。当你想改变这些属性之一时，你调用一个可以改变设备环境中属性的函数，以后针对该设备环境的TextOut调用将使用改变后的属性。

4.3.1 获取设备环境的句柄

Windows提供了几种获取设备环境句柄的方法。如果在处理一条消息时获取了设备环境句柄，应该在退出窗口函数之前释放它(或者删除它)。一旦释放了句柄，它就不再有效了。

最常用的获取设备环境句柄并释放它的方法是在处理WM_PAINT消息时，使用BeginPaint和EndPaint调用:

```
hdc = BeginPaint (hwnd, &ps) ;
[其它程序行]
EndPaint (hwnd, &ps) ;
```

变量ps是类型为PAINTSTRUCT的结构，该结构的hdc字段是BeginPaint返回的设备环境句柄。PAINTSTRUCT结构又包含一个名为rcPaint的RECT(矩形)结构，rcPaint定义一个包围窗口客户区的无效范围的矩形。使用从BeginPaint获得的设备环境句柄，只能在这个

区域内绘图。BeginPaint 调用使该区域有效。

Windows 程序还可以在处理非 WM_PAINT 消息时获取设备环境句柄:

```
hdc = GetDC (hwnd) ;
[其它程序行]
ReleaseDC (hwnd, hdc) ;
```

这个设备环境适用其句柄为 hwnd 的窗口的客户区。这些调用与 BeginPaint 和 EndPaint 的组合之间的基本区别是，利用从 GetDC 返回的句柄可以在整个客户区上绘图。当然，GetDC 和 ReleaseDC 不使客户区中任何可能的无效区域变成有效。

Windows 程序还可以获取适用于整个窗口(而不仅限于窗口的客户区)的设备环境句柄:

```
hdc = GetWindowDC (hwnd) ;
[其它程序行]
ReleaseDC (hwnd, hdc) ;
```

这个设备环境除了客户区之外，还包括窗口的标题条、菜单、滚动条和框架(frame)。GetWindowDC 函数很少使用，如果想尝试用一用它,则必须捕获 WM_NCPAINT(“非客户绘制”)消息，该消息阻止 Windows 在窗口的非客户区上绘图。

BeginPaint 、GetDC 和 GetWindowDC 获得的设备环境都与视频显示器上的某个特定窗口相关。获取一个设备环境句柄的另一个更通用的函数是 CreateDC:

```
hdc = CreateDC (pszDriver, pszDevice, pszOutput, pData) ;
[其它程序行]
DeleteDC (hdc) ;
```

例如，你可以通过下面的调用来获取整个屏幕的设备环境句柄:

```
hdc = CreateDC ("DISPLAY", NULL, NULL) ;
```

在窗口之外写东西一般是不合适的，但对于一些不同寻常的应用程序来说，这样做很方便(你还可通过在调用 GetDC 时使用一个 NULL 参数，来获取整个屏幕的设备环境句柄，不过这一事实并没有被载入文档中)。在第十五章中，我们将使用 CreateDC 函数来获取一个打印机设备环境的句柄。

有时你只是需要获取有关一个设备环境的一些信息而并不进行任何绘画，在这种情况下,你可以使用 CreateIC 来获取一个“信息环境”的句柄，其参数与 CreateDC 函数相同，例如:

```
hdcInfo = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
[其它程序行]
DeleteDC (hdcInfo) ;
```

你不能用这个信息环境句柄往设备上写东西。

当使用位图时，获取一个“内存设备环境”有时是有用的:

```
hdcMem = CreateCompatibleDC (hdc) ;
[其它程序行]
DeleteDC (hdcMem) ;
```

这是一个相当抽象的概念。但是，大体上，你所做的就是将一个位图选入内存设备环境中，然后使用 GDI 函数调用在位图上绘画。我们将在这一章稍后进行这方面的讨论，并使用第 10 章的 GRAPMENU 程序中的技术。

我前面已经提到过，元文件是一些 GDI 调用的集合，以二进制形式编码。你可以通过获取一个元文件设备环境来创建一个元文件:

```
hdcMeta = CreateMetaFile (pszFilename) ;
[其它程序行]
DeleteDC (hdcMeta) ;
```

在元文件设备环境有效期间，任意用 hdcMeta 所做的 GDI 调用都变成元文件的一部分而不会被显示。在调用 CloseMetaFile 之后，设备环境句柄变为无效，函数返回一个指向元文件（hmf)的句柄.

4.3.2 获取设备环境信息

一个设备环境通常是指一个物理显示设备，如视频显示器和打印机。通常，你需要获取有关该设备的信息，包括显示器的大小（单位为象素或者物理度量）和色彩容量。你可以通过调用 GetDeviceCaps(“获取设备能力”)函数来获取这些信息:

```
nValue = GetDeviceCaps (hdc, iIndex) ;
```

其中，参数 iIndex 取值为 WINDOWS.H 中定义的 28 个标识符之一。例如， iIndex 为 HORZRES 时将导致 GetDeviceCaps 返回设备的宽度(以象素为单位)； iIndex 为 VERTRES 时将导致 GetDeviceCaps 返回设备的高度（单位为象素）。如果 hdc 是屏幕设备环境的句柄，则 GetDeviceCaps 返回打印机显示区域的高度和宽度，它们也是以象素为单位的。

还可以使用 GetDeviceCaps 来确定设备处理不同类型图形的能力，这对于视频显示器并不是很重要，但是对于打印设备却是非常重要。例如，大多数绘图仪不能画位图图象，GetDeviceCaps 就可以将这一情况告诉你。

4.3.3 DEVCAPS1 程序

图 4-1 所示的 DEVCAPS1 程序显示了，以一个视频显示器的设备环境为参数时，可以从 GetDeviceCaps 函数中获得的部分信息（该程序的另一个扩充版本 DEVCAPS2 将在第十五章中给出，用于获取打印机信息）。

DEVCAPS1.MAK

```
#------------------------
```

```
# DEVCAPS1.MAK make file
#----------------------

devcaps1.exe : devcaps1.obj
     $(LINKER) $(GUIFLAGS) -OUT:devcaps1.exe devcaps1.obj $(GUILIBS)

devcaps1.obj : devcaps1.c
     $(CC) $(CFLAGS) devcaps1.c
```

DEVCAPS1.C

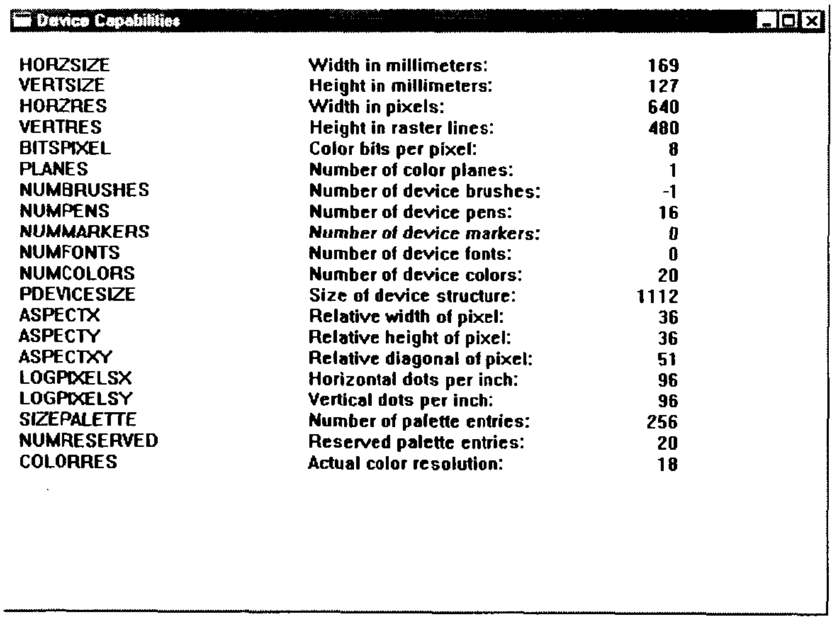
```
/*---------------------------------------------------------
   DEVCAPS1.C -- Device Capabilities Display Program No. 1
                 (c) Charles Petzold, 1996
  ---------------------------------------------------------*/
#include <windows.h>
#include <string.h>

#define NUMLINES ((int) (sizeof devcaps / sizeof devcaps [0]))

struct
     {
     int  iIndex ;
     char *szLabel ;
     char *szDesc ;
     }
     devcaps [] =
          {
          HORZSIZE,      "HORZSIZE",     "Width in millimeters:",
          VERTSIZE,      "VERTSIZE",     "Height in millimeters:",
          HORZRES,       "HORZRES",      "Width in pixels:",
          VERTRES,       "VERTRES",      "Height in raster lines:",
          BITSPIXEL,     "BITSPIXEL",    "Color bits per pixel:",
          PLANES,        "PLANES",       "Number of color planes:",
          NUMBRUSHES,    "NUMBRUSHES",   "Number of device brushes:",
          NUMPENS,       "NUMPENS",      "Number of device pens:",
          NUMMARKERS,    "NUMMARKERS",   "Number of device markers:",
          NUMFONTS,      "NUMFONTS",     "Number of device fonts:",
          NUMCOLORS,     "NUMCOLORS",    "Number of device colors:",
          PDEVICESIZE,   "PDEVICESIZE",  "Size of device structure:",
          ASPECTX,       "ASPECTX",      "Relative width of pixel:",
          ASPECTY,       "ASPECTY",      "Relative height of pixel:",
          ASPECTXY,      "ASPECTXY",     "Relative diagonal of pixel:",
          LOGPIXELSX,    "LOGPIXELSX",   "Horizontal dots per inch:",
          LOGPIXELSY,    "LOGPIXELSY",   "Vertical dots per inch:",
          SIZEPALETTE,   "SIZEPALETTE",  "Number of palette entries:",
          NUMRESERVED,   "NUMRESERVED",  "Reserved palette entries:",
          COLORRES,      "COLORRES",     "Actual color resolution:"
          } ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "DevCaps1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Device Capabilities",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

          switch (iMsg)
          {
```

```
        case WM_CREATE :
             hdc = GetDC (hwnd) ;

             GetTextMetrics (hdc, &tm) ;
             cxChar = tm.tmAveCharWidth ;

             cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
             cyChar = tm.tmHeight + tm.tmExternalLeading ;

             ReleaseDC (hwnd, hdc) ;
             return 0 ;

        case WM_PAINT :
             hdc = BeginPaint (hwnd, &ps) ;

             for (i = 0 ; i < NUMLINES ; i++)
                  {
                  TextOut (hdc, cxChar, cyChar * (1 + i),
                           devcaps[i].szLabel,
                           strlen (devcaps[i].szLabel)) ;

                  TextOut (hdc, cxChar + 22 * cxCaps, cyChar * (1 + i),
                           devcaps[i].szDesc,
                           strlen (devcaps[i].szDesc)) ;

                  SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

                  TextOut (hdc, cxChar + 22 * cxCaps + 40 * cxChar,
                           cyChar * (1 + i), szBuffer,
                           wsprintf (szBuffer, "%5d",
                               GetDeviceCaps (hdc, devcaps[i].iIndex))) ;

                  SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                  }

             EndPaint (hwnd, &ps) ;
             return 0 ;

        case WM_DESTROY :
             PostQuitMessage (0) ;
             return 0 ;
        }

   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
   }
```

图 4-1 DEVCAPS1程序

可以看到，这个程序非常类似于第3章中所示的SYSMETS1程序。为了保持代码的短小，我没有包括滚动条，因为我知道信息可以在一屏上显示出来。在我的256色VGA上的显示结果如图4-2所示。


Device Capabilities

HORZSIZE	Width in millimeters:	169
VERTSIZE	Height in millimeters:	127
HORZRES	Width in pixels:	640
VERTRES	Height in raster lines:	480
BITSPIXEL	Color bits per pixel:	8
PLANES	Number of color planes:	1
NUMBRUSHES	Number of device brushes:	-1
NUMPENS	Number of device pens:	16
NUMMARKERS	Number of device markers:	0
NUMFONTS	Number of device fonts:	0
NUMCOLORS	Number of device colors:	20
PDEVICESIZE	Size of device structure:	1112
ASPECTX	Relative width of pixel:	36
ASPECTY	Relative height of pixel:	36
ASPECTXY	Relative diagonal of pixel:	51
LOGPIXELSX	Horizontal dots per inch:	96
LOGPIXELSY	Vertical dots per inch:	96
SIZEPALETTE	Number of palette entries:	256
NUMRESERVED	Reserved palette entries:	20
COLORRES	Actual color resolution:	18

图 4-2 一个256色VGA上的DEVCAPS1显示

4.3.4 设备的大小

在 Windows 程序能够从 GetDeviceCaps 获得的关于视频设备的信息中，最重要的是显示器的大小（两种单位:毫米和象素）和显示器象素的纵横比，这些尺寸将有助于对显示图象进行放缩。

HORZSIZE 和 VERTSIZE 的值是显示区域的宽度和高度，以毫米为单位。当然，Windows 驱动程序并不知道用户连接到视频适配器上的显示器的真实尺寸，这些数值是根据适配器的标准显示尺寸给出的。

HORZRES 和 VERTRES 的值是显示区域的宽度和高度，以象素为单位。对于一个视频显示器的设备环境，它们与 GetSystemMetrics 返回的值相同。使用这些值，加上 HORZSIZE 和 VERTSIZE，你可以获得以每毫米的象素表示的设备分辨率。如果你知道一英寸有 25.4 毫米(mm)，那么你也可以获得以每英寸的象素表示的分辨率。

ASPECTX 、 ASPECTY 和 ASPECTXY 的值分别是每个象素的相对宽、高和对角线长度。 ACPECTXY 等于 ASPECTX 和 ASPECTY 的平方和的平方根。

LOGPIXELSX 和 LOGPIXELSY 的值分别是每个水平和垂直“逻辑英寸”中的象素数。对于一个视频显示器来说，一逻辑英寸并不等于真正的一英寸(25.4mm)，只要用 HORZSIZE 、 VERTSIZE 、 HORZRES 和 VERTRES 的值进行简单的运算就可以知道。LOGPIXELSX 和 LOGPIXELSY 的含义需要解释一下。你可能已经注意到了，许多的 Windows 字处理程序显示的标尺并不正确：如果你测量一下 VGA 监视器上显示的标尺，就可以发现，它的 1 英寸实际上要比 1.5 英寸还要多。这些程序的标尺使用的正是 LOGPIXELSX 和 LOGPIXELSY 的值，如果程序使用真实的物理尺度，则普通的 10 点或者 12 点文本将会小到几乎难以辨认。这些逻辑尺寸的实际效果是放大了显示，使得显示的

文本能够有一个适当的尺寸。当我们开始使用文本时，还会遇到这个问题，它只影响视频显示；对于打印机，GetDeviceCaps 返回的所有尺寸都是一致的。

4.3.5 关于色彩

彩色显示要求每个象素不止一位，位数越多，色彩越多，或者更具体地说，可以同时显示的不同色彩的数目等于 2 的位数次方。16 色的视频适配器要求每个象素 4 位，这些位按色彩平面来组织——一个红色平面、一个绿色平面、一个兰色平面和一个强度平面。每个象素占用 8 、16 或者 24 位的适配器只有一个色彩平面，其中，若干相邻位代表每个象素的色彩。

GetDeviceCaps 使程序员可以确定视频适配器的存储组织以及它能够表示的色彩数目，这个调用返回色彩平面的数目:

```
iPlanes = GetDeviceCaps(hdc, PLANES) ;
```

下面的调用返回每个象素的色彩位数:

```
iBitsPixel = GetDeviceCaps(hdc, BITSPIXEL) ;
```

大多数彩色图形显示设备要么使用多个色彩平面要么每象素有多个色彩位，但是不能同时二者兼用；换句话说，这两个调用必有一个返回 1 。视频适配器能够表示的色彩数可以用如下公式来计算出来:

```
iColor = 1<<(iPlanes * iBitPixel) ;
```

这个值与用 NUMCOLORS 参数得到的色彩数值可能一样，也可能不一样：

```
iColors = GetDeviceCaps(hdc, NUMCOLORS) ;
```

例如，对于大部分绘图仪，这两个值都不相同。对于绘图仪，PLANES 和 BITPIXEL 的值都等于 1 ，NUMCOLORS 的值反映绘图仪拥有的彩色画笔数。对于单色设备，GetDeviceCaps 对于 NUMCOLORS 参数返回 2 。

更重要的是，这两个值对于支持可以加载色彩调色板的 256 色视频适配器也可以是不同的。以 NUMCOLORS 为参数时，GetDeviceCaps 返回由 Windows 保留的色彩数目，值为 20 ，剩下的 236 种色彩可以由 Windows 程序用调色板管理器设置。

Windows 使用一个无符号长整数(32 位)来表示一种色彩。色彩的数据类型被称为 COLORREF ，最低的三个字节分别指定红、绿和蓝，范围从 0 到 255 ，如图 4-3 中所示。这将有可能产生一个 2(或者大约一千六百万)种色彩的调色板。

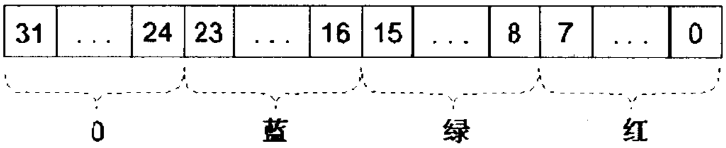


图 4-3 32位色彩值

这个无符号长整数常常被称为一个“ RGB 色彩”。Windows 头文件提供了几种使用 RGB 色彩值的宏。RGB 宏要求三个参数分别代表红、绿和蓝值，然后将它们组合为一个

无符号长整数:

```
#define RGB(r, g, b) ((COLORREF)(((BYTE)(R) | \
                    ((WORD)(g) << 8)) | \
                    (((DWORD)(BYTE)(b)) << 16)))
```

因此，值:

```
RGB(255, 0, 255)
```

实际上是 0x00FF00FF，这是紫色的 RGB 色彩值。当所有三个参数被设置为 0 时，色彩为黑色；当所有参数被设置为 255 时，色彩为白色。GetRValue、GetGValue 和 GetBValue 宏从一个无符号长整数 RGB 色彩值中抽取出无符号字节的原色值。当你在使用返回 RGB 色彩值的函数时，这些宏有时会很方便。

由 GetDeviceCaps 返回的色彩数是设备可以显示的纯色数。Windows 可以通过抖动（利用了由多种色彩的象素组成的象素图案）来表示纯色之外的色彩。并非所有不同的红、绿、蓝组合都产生不同的抖动图案。例如，在十六色 VGA 上，红、绿、蓝值一般必须增加 4 才能产生出一个不同的抖动图案。因此，对于这些适配器，可以得到 2^{18}（262,144）种抖动色彩。

可以调用 GetNearestColor 来决定与某一色彩最接近的纯色。

(rgbPureColor = GetNearestColor(hDc, rgbColor);

4.3.6 设备环境属性

前面已经提到过，Windows 使用设备环境来保存控制 GDI 函数在显示器上如何操作的“属性”。例如，在用 TextOut 函数显示文本时，程序员不必指定文本的色彩和字体，Windows 从设备环境获取这个信息。

当程序获取一个设备环境的句柄时，Windows 创建一个设备环境，其中所有的属性都被设置为缺省值。设备环境属性如下表所示，程序可以改变或者获取任何一种属性。

设备环境属性	缺省值	修改该值的函数	获取该值的函数
映射模式	MM_TEXT	SetMapMode	GetMapMode
窗口原点	(0, 0)	SetWindowOrgEx OffsetWindowOrgEx	GetWindowOrgEx
视口原点	(0, 0)	SetViewportOrgEx OffsetViewportOrgEx	GetViewportOrgEx
窗口范围	(1, 1)	SetWindowExtEx SetMapMode ScaleWindowExtEx	GetWindowExtEx
视口范围	(1, 1)	SetViewportExtEx SetMapMode ScaleViewportExtEx	GetViewportExtEx
画笔	BLACK_PEN	SelectObject	SelectObject
画刷	WHITE_BRUSH	SelectObject	SelectObject

续表

设备环境属性	缺省值	修改该值的函数	获取该值的函数
字体	SYSTEM_FONT	SelectObject	SelectObject
位图	无	SelectObject	SelectObject
当前画笔位置	(0, 0)	MoveToEx LineTo PolylineTo PolyBezierTo	GetCurrentPositionEx
背景模式	OPAQUE	SetBkMode	GetBkMode
背景色彩	白	SetBkColor	GetBkColor
文本色	黑	SetTextColor	GetTextColor
绘图模式	R2_COPYPEN	SetROP2	GetROP2
伸缩模式	BLACKONWHITE	SetStretchBltMode	GetStrechBltMode
多边形填充模式	ALTERNATE	SetPolyFillMode	GetPolyFillMode
字符间距	0	SetTextCharacterExtra	GetTextCharacterExtra
画笔原点	屏幕坐标（0,0）	SetBrushOrgEx	GetBrushOrgEx
剪裁区域	无	SelectObject SelectClipRgn IntersectClipRect OffsetClipRgn ExcludeClipRgn SelectClipPath	GetClipBox

4.3.7 保存设备环境

在整个这一章中，你将会遇到许多改变设备环境属性的函数。通常，在你调用 GetDC 或者 BeginPaint 时， Windows 用缺省值创建一个新的设备环境，你对属性所做的一切改变在设备环境用 ReleaseDC 或者 EndPaint 调用释放时，都会丢失。如果你的程序需要使用非缺省的设备环境属性，则你必须在每次获取设备环境句柄时初始化设备环境:

```
case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;
    [初始化设备环境属性]
    [绘制窗口的客户区]
    EndPaint (hwnd, &ps) ;
    return 0 ;
```

虽然在通常情况下这种方法已经很令人满意了，但是你还可能想要在释放设备环境之后，仍然保存程序中对设备环境属性所做的改变，以便在下一次调用 GetDC 或者 BeginPaint 时它们仍然能够起作用。为此，可在登录窗口类时，将 CS_OWNDC 标志包含为窗口类的一部分:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC ;
```

现在，基于这个窗口类所创建的每个窗口都将拥有自己的设备环境，它一直存在，直到窗口被删除。如果使用了 CS_OWNDC 风格，就只需初始化设备环境一次，可以在处理 WM_CREATE 消息期间完成这一操作:

```
case WM_CREATE:
    hdc = GetDC (hwnd) ;
    [初始化设备环境属性]
    ReleaseDC (hwnd, hdc) ;
```

这些属性在被改变之前一直有效。

CS_OWNDC 风格只影响 GetDC 和 BeginPaint 获得的设备环境，不影响其他函数(如 GetWindowDC)获得的设备环境。CS_OWNDC 风格并不是没有代价的：Windows 需要为每个采用这种风格创建的窗口花费大约 800 个字节来保存设备环境；即使用了 CS_OWNDC，你仍然应该在退出窗口过程之前释放设备环境。

你也可以使用 CS_CLASSDC 风格:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_CLASSDC;
```

这种风格导致每个窗口类拥有自己的设备环境，环境由所有在该类上创建的窗口共享。一般来说，CS_CLASSDC 风格的设备环境比 CS_OWNDC 风格要难用一些，因为你对设备环境所做的任何改变都会影响到基于同一窗口类的所有窗口，这也许会导致一些奇怪的效果。

某些情况下，你可能想改变某些设备环境属性，用改变后的属性进行绘图，然后恢复原来的设备环境。要简化这一过程，可以通过如下调用来保存设备环境的状态:

```
iSavedID = SaveDC (hdc) ;
```

现在，可以改变一些属性，在想要回到调用 SaveDC 前存在的设备环境时，调用:

```
RestoreDC (hdc, iSaveID) ;
```

你可以在调用 RestoreDC 之前调用 SaveDC 数次。如果想恢复到最后一次调用 SaveDC 前存在的设备环境，可调用:

```
RestoreDC( hdc, -1) ;
```

4.4 画 线

理论上，只要提供一个 SetPixel(有时，还需要一个 GetPixel)函数，就可以使用一个设备驱动程序来绘制一切东西了。别的一切可以使用更高级一些的函数来处理，可能是在 GDI 模块中实现，或者甚至是在你的应用程序代码中实现。例如，画线时，只需多次地调用“写象素”例程，并适当地调整 x 和 y 坐标。

在实践中，也的确可以仅仅使用“写象素”和“读象素”例程来做所想做的事——只要你不在乎苦等结果出来。对于一个图形系统来说，在设备驱动程序的层次上完成画线和其它复杂的图形操作，效率要高得多。因为设备驱动程序可以通过自己优化后的代码来完成这些操作；并且，随着视频显示技术越来越复杂，适配器上将会包含图形协处理器，使得视频硬件能够自己绘图。

Windows GDI 当然包含了 SetPixel 和 GetPixel 函数。虽然我们将在第 6 章中的 CONNECT 程序中使用 SetPixel 函数，但在实际的图形程序中，很少使用这些函数。对于多数情况，线必须被认为是最低层的矢量图元。

Windows 可以画直线、椭圆线(在椭圆圆周线上的一条曲线)和 Bezier 样条。Windows 95 支持的七个画线函数是 LineTo(直线)、Polyline 和 PolylineTo(一系列相连的直线)、PolyPolyline(多组相连的线)、Arc(椭圆线)、PolyBezier 和 PolyBezierTo(Windows NT GDI 支持另外的三个画线函数—— ArcTo 、AngleArc 和 PolyDraw ，这三个函数 Windows 95 不支持。)设备环境的五种属性影响这些函数所画的线，这些属性是：当前画笔的位置（影响 LineTo 、PolylineTo 和 PolyBezierTo ）、画笔、背景方式（对于非实心画笔)、背景色彩（对于 OPAQUE 背景方式）和绘图模式。

由于我很快会讨论到的原因，我还将在这一节中介绍 Rectangle 、Ellipse 、RoundRect 、Chord 和 Pie 函数，虽然这些函数既画线又填充封闭区域。

LineTo 是不包含绘制对象全部尺寸数据的 GDI 函数之一，它从设备环境中定义的当前画笔位置开始，一直画到函数中所指定的点为止。在缺省设备环境里，当前画笔位置最初设置在点(0 ， 0)。如果在调用 LineTo 函数之前不先设置当前的画笔位置（或者改变窗口或者视口的原点），那么它将从客户区的左上角开始画线。

要在逻辑点(xStart ， yStart)到(xEnd ， yEnd)之间画一条直线，应该先用 MoveToEx 函数将画笔位置设置到点(xStart ， yStart):

```
MoveToEx (hdc, xStart, yStart, &pt) ;
```

这里，pt 是一个类型为 POINT 的结构，在 Windows 头文件中如下定义:

```
typedef struct tag POINT
    {
    long x ;
    long y ;
    } POINT ;
```

MoveToEx 不画任何对象，它只是改变当前画笔位置，以前的当前位置被保存在 POINT 结构中，然后你就可以使用 LineTo 来画线了:

```
LineTo (hdc, xEnd, yEnd) ;
```

这条语句从当前画笔位置开始画一条直线到点(xEnd, yEnd)(不包含点 xEnd ， yEnd)。在调用 LineTo 之后，当前画笔位置变成(xEnd, yEnd)。

一点历史：在 Windows 的 16 位版本中，用来改变当前位置的函数是 MoveTo ，该函

数只调整三个参数——设备环境句柄、x 和 y 坐标。函数通过两个 16 位数拼成的 32 位无符号长整数返回先前的当前位置。然而，在 Windows 的 32 位版本中(包括 Windows NT 和 Windows 95)，坐标是 32 位的数值，而 C 的 32 位版本中又没有定义 64 位的整数数据类型，因此这种改变要求 MoveTo 被替换为 MoveToEx 。这种改变是必要，尽管在实际的程序中，由 MoveTo 返回的值几乎从不被使用。

一条好消息：如果你不需要以前的当前位置——几乎总是这样——那么你可以简单地将 MoveToEx 的最后一个参数设置为 NULL 。实际上，为了将一些已经存在的 16 位代码转换到 Windows 95 中，你也许想定义一个宏，如下所示:

```
#define MoveTo(hdc, x, y) MoveToEx(hdc, x, y, NULL)
```

在以后章节的几个程序中，我使用了这个宏。

一条坏消息：尽管 Windows 95 中的坐标值看起来是 32 位的，实际上却只用到了低 16 位，坐标值实际上被限制在－32,768 到 32,767 之间。

你可以通过调用:

```
GetCurrentPositionEx (hdc, &pt) ;
```

来获得当前的画笔位置。

下面的代码从窗口的左上角开始，在客户区中画一个网格，线与线之间相隔 100 个象素，其中 hwnd 是窗口的句柄，hdc 是设备句柄，而 x 和 y 是整数:

```
GetClientRect (hwnd, &rect) ;
for (x = 0 ; x < rect.right ; x += 100)
     {
     MoveToEx (hdc, x, 0, NULL) ;
     LineTo (hdc, x, rect.bottom) ;
     }

for (y = 0 ; y < rect.bottom ; y += 100)
     {
     MoveToEx (hdc, x, 0, NULL) ;
     LineTo (hdc, x, rect.right, y) ;
     }
```

虽然用两个函数来画一条直线显得有些麻烦，但是在希望画一组相连的直线时，当前画笔位置属性又会变得很有用。例如，你可能想定义一个包含 5 个点(10 个值)的数组，来画一个矩形的边界框:

```
POINT pt[5] = {100 , 100 , 200 , 100 , 200 , 200 , 100 , 200 , 100 , 100} ;
```

注意，最后一个点与第一个点相同。现在，只需要使用 MoveToEx 移到第一个点，并对后面的点使用 LineTo:

```
MoveToEx (hdc, pt[0].x, pt[0].y, NULL) ;
for (i=1; i<5; i++)
```

```
LineTo (hdc, pt[1].x, pt[1].y) ;
```

由于 LineTo 从当前位置画到（但不包括） LineTo 函数中给出的点，所以这段代码没在任何坐标处画两次。虽然在显示器上多输出几次不存在问题，但是在绘图仪上或者在其它绘图方式（下面马上会讲到)下，视觉效果就不太好了。

当你要将数组中的点连接成线时，使用 Polyline 函数要简单得多。下面这条语句画出与上面一段代码相同的矩形:

```
Polyline (hdc, &pt, 5) ;
```

最后一个参数是点的数目。我们还可以使用(sizeof(pt)/sizeof(POINT))来表示这个值。Polyline 与一个 MoveToEx 函数后面加几个 LineTo 函数的效果相同，但是， Polyline 既不使用也不改变当前位置。 PolylineTo 有些不同，这个函数使用当前位置作为开始点，并将当前位置设置为最后一根线的终点。下面的代码画出与上面所示一样的矩形:

```
MoveToEx (hdc, pt[0].x, pt[0].y, NULL) ;
PolylineTo (hdc, pt + 1, 4) ;
```

你可以对几条线使用 Polyline 和 PolylineTo ，不过，当绘制由几百甚至几千条线组成的复杂曲线时，这些函数才是真正的极其有用。例如，画正弦波就是这样的，图 4-4 所示的 SINEWAVE 程序显示了如何做到这一点。

SINEWAVE.MAK

```
#------------------------
# SINEWAVE.MAK make file
#------------------------

sinewave.exe : sinewave.obj
     $(LINKER) $(GUIFLAGS) -OUT:sinewave.exe sinewave.obj $(GUILIBS)

sinewave.obj : sinewave.c
     $(CC) $(CFLAGS) sinewave.c
```

SINEWAVE.C

```
/*-----------------------------------------
   SINEWAVE.C -- Sine Wave Using Polyline
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <math.h>

#define NUM    1000
#define TWOPI  (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
```

```
     {
     static char szAppName[] = "SineWave" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Sine Wave Using Polyline",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static int  cxClient, cyClient ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     POINT       pt [NUM] ;

     switch (iMsg)
          {
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
```

```
          MoveToEx (hdc, 0,        cyClient / 2, NULL) ;
          LineTo   (hdc, cxClient, cyClient / 2) ;

          for (i = 0 ; i < NUM ; i++)
               {
               pt[i].x = i * cxClient / NUM ;
               pt[i].y = (int) (cyClient / 2 *
                              (1 - sin (TWOPI * i / NUM))) ;
               }

          Polyline (hdc, pt, NUM) ;

          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 4-4 SINEWAVE程序

这个程序有一个含有1000个POINT结构的数组。随着for循环从0增加到999，结构的x成员被设置为从0递增到数值cxClient。结构的y成员被设置为一个周期的正弦曲线值，并被放大以充满客户区。整个曲线的绘制仅仅使用了一个Polyline调用。因为Polyline函数是在设备驱动器级别上实现的，因此它要比调用1000次LineTo快得多，结果如图4-5中所示。

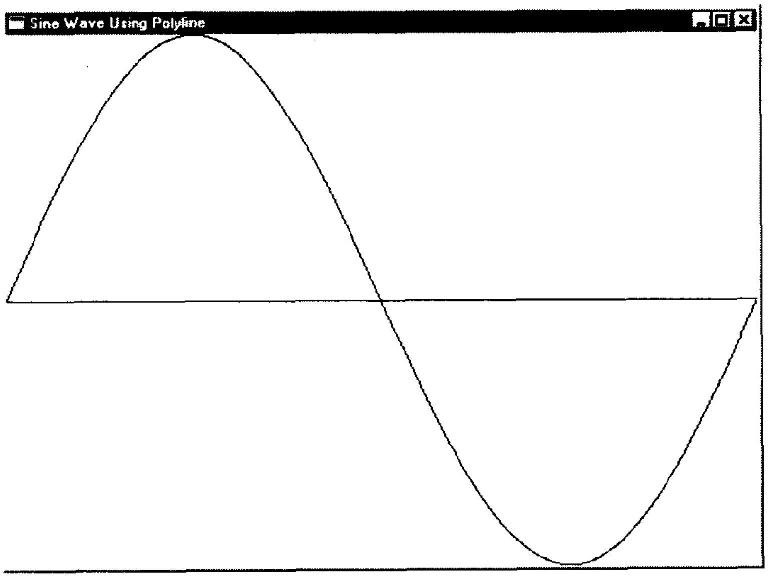


图 4-5 SINEWAVE显示

4.4.1 边界框

下面我想讨论的是Arc函数，它绘制椭圆曲线。然而，如果不先讨论一下Ellipse函数，那么Arc函数将难以理解；而如果不先讨论Rectangle函数，那么Ellipse函数又将难以理

解；而如果讨论 Ellipse 和 Rectangle 函数，那么我又会讨论 RoundRect 、 Chord 和 Pie 函数。

问题在于， Rectangle 、 Ellipse 、 RoundRect 、 Chord 和 Pie 函数严格来说不是画线函数。没错，这些函数是在画线，但它们同时又在使用当前的区域填充刷子填充一个封闭区域。这个刷子缺省为白色，因此当你第一次使用这些函数时，你可能不会注意到它们不只是画线。严格地说，这些函数属于后面关于“填充区域”的小节，不过，我还是在这里讨论它们。

上面提到的函数有一个共性，即它们都是基于一个矩形边界框的。你定义一个包含该对象的框，即“边界框(bounding box)”； Windows 就在这个框内画出该对象。

这些函数中最简单的就是画一个矩形：

```
Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
```

点(xLeft, yTop)是矩形的左上角，(xRight, yBottom)是矩形的右下角；两个点都以逻辑单位来表示。用函数 Rectangle 画出的图形如图 4-6 中所示，矩形的边总是平行于显示器的水平和垂直边。

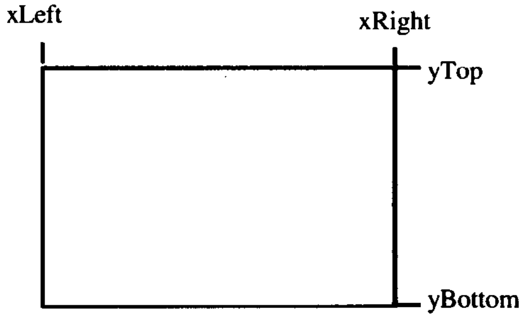


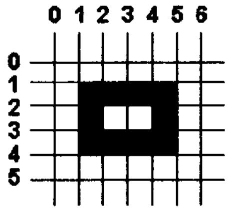
图 4-6 使用Rectangle函数画出的一个图形

以前编过图形程序的程序员熟悉象素偏差的问题。有些图形系统画出的图形包含右坐标和底坐标，而有些则只画到(而不包含)右坐标和底坐标。 Windows 采用后一种方法，不过有一种更简单的方法来考虑这个问题。

考虑下面的函数调用：

```
Rectangle (hdc, 1 , 1 , 5 , 4)
```

上面我们提到， Windows 在边界框内画图。可以将显示器设想成一个网格，其中，每个象素都在一个网格单元内。边界框画在网格上，然后在边界框内画矩形，下面说明了图形画出来时的样子：



将矩形和客户区左上角分开的区域有 1 个象素宽。 Windows 使用当前刷子来着色矩形内的两个象素。

你知道了如何画矩形，也就知道了如何画椭圆，因为它们使用的参数都是相同的:

```
Ellipse (hdc, xLeft, yTop, xRight, yBottom) ;
```

用 Ellipse 函数画出的图形如图 4-7 所示(加上了虚线构成的边界框)。

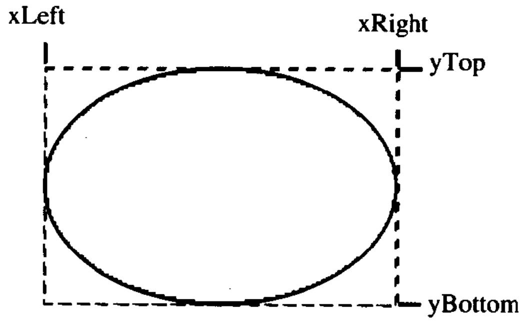


图 4-7　用Ellipse函数画出的图形

画圆角矩形的函数使用与函数 Rectangle 及 Ellipse 函数相同的边界框，还包含另外两个参数:

```
RoundRect(hdc, xLeft, yTop, xRight, yBottom,
          xCornerEllipse, yCornerEllipse) ;
```

用这个函数画出的图形如图 4-8 所示。

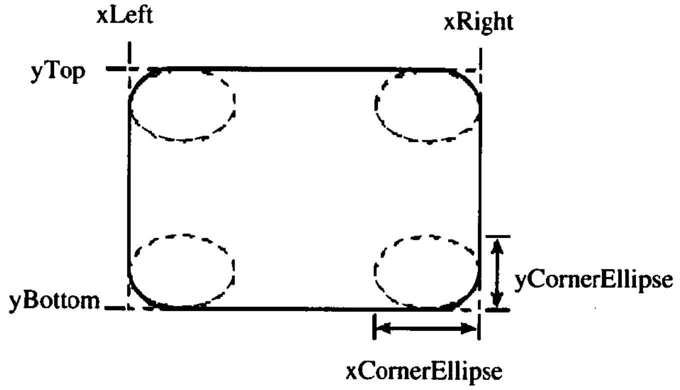


图 4-8　用函数RondRect画出的图形

Windows 使用一个小椭圆来画圆角，这个椭圆的宽为 xCornerEllipse ，高为 yCornerEllipse ，这两个值都是以逻辑单位表示的。可以想象这个小椭圆分为了四个部分，一个象限一个，每个刚好用在矩形的一个角上。xCornerEllipse 和 yCornerEllipse 的值越大，角就越明显。如果 xCornerEllipse 等于 xLeft 与 xRight 的差，且 yCornerEllipse 等于 yTop 与 yBottom 的差，那么 RoundRect 函数将画出一个椭圆。

在绘制图 4-8 所示的圆角矩形时，用了下面的公式来计算 xCornerEllipse 与 yCornerEllipse 的值:

```
xCornerEllipse = (xRigth - xLeft) / 4 ;
yCornerEllipse = (yBottom - yTop) / 4 ;
```

这是一种简单的方法，但是结果看起来有点不对劲儿，因为角的弯曲部分在矩形长的一边要大些。要矫正这一问题，你可以让 xConerEllipse 与 yCornerEllipse 的值相等。

Arc 、 Chord 和 Pie 函数都使用相同的参数:

```
Arc (hdc, xLeft, yTop, xRight, yBottom,
      xStart, yStart, xEnd, yEnd) ;
Chord (hdc, xLeft, yTop, xRight, yBottom,
      xStart, yStart, xEnd, yEnd) ;
Pie (hdc, xLeft, yTop, xRight, yBottom,
      xStart, yStart, xEnd, yEnd) ;
```

用 Arc 函数画出的线如图 4-9 所示；用 Chord 和 Pie 函数画出的线分别如图 4-10 和 4-11 所示。 Windows 用一条假想的线将(xStart, yStart)与椭圆的中心连接，从该线与边界框的交点开始， Windows 按反时针方向,沿着椭圆画一条弧。 Windows 还用另一条假想的线将(xEnd, yEnd)与椭圆的中心连接，在该线与边界框的交点处， Windows 停止画弧。

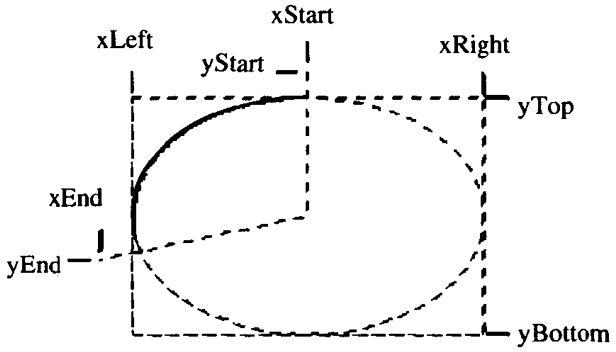


图 4-9 Arc函数画出的线

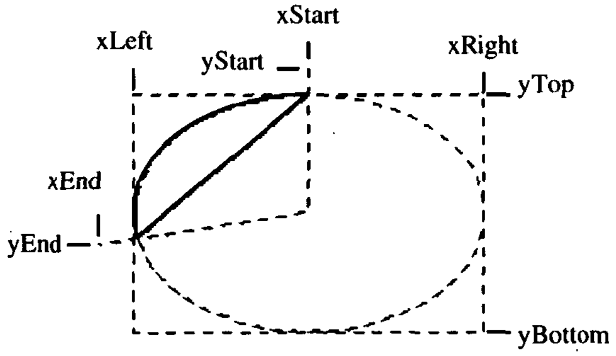


图 4-10 Chord函数画出的图形

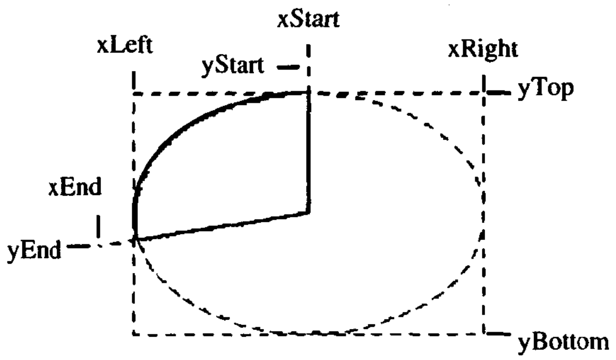


图 4-11 Pie函数画出的图形

对于 Arc 函数，这样就结束了。因为弧只是一条椭圆形的线而已，而不是一个填充区域。对于 Chord 函数， Windows 连接弧线的端点。而对于 Pie 函数， Windows 将弧的两个端点与椭圆的中心相连接。弦与饼形图的内部以当前刷子填充。

你可能不太明白在 Arc 、 Chord 和 Pie 函数中开始和结束位置的用法，为什么不简单地在椭圆的周线上指定开始和结束点呢？是的，你可以这么做，但是你将不得不算出这些点。 Windows 的方法在不要求这种精确性的条件下，却完成了相同的工作。

图 4-12 中所示的 LINEDEMO 程序画一个矩形、一个椭圆、一个圆角矩形和两条线段，不过不是按这一顺序。程序表明了定义封闭区域的函数实际上对这些区域进行了填充，因为在椭圆后面的线被遮住了，结果如图 4-13 中所示。

LINEDEMO.MAK

```
#-----------------------
# LINEDEMO.MAK make file
#-----------------------

linedemo.exe : linedemo.obj
     $(LINKER) $(GUIFLAGS) -OUT:linedemo.exe linedemo.obj $(GUILIBS)

linedemo.obj : linedemo.c
     $(CC) $(CFLAGS) linedemo.c
```

LINEDEMO.C

```
/*--------------------------------------------------
   LINEDEMO.C -- Line-Drawing Demonstration Program
                 (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "LineDemo" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
```

```
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Line Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static int  cxClient, cyClient ;
     HDC         hdc ;
     PAINTSTRUCT ps ;

     switch (iMsg)
          {
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               Rectangle (hdc,     cxClient / 8,     cyClient / 8,
                               7 * cxClient / 8, 7 * cyClient / 8) ;

               MoveToEx  (hdc,        0,        0, NULL) ;
               LineTo    (hdc, cxClient, cyClient) ;

               MoveToEx  (hdc,        0, cyClient, NULL) ;
               LineTo    (hdc, cxClient,        0) ;

               Ellipse   (hdc,     cxClient / 8,     cyClient / 8,
                               7 * cxClient / 8, 7 * cyClient / 8) ;

               RoundRect (hdc,     cxClient / 4,     cyClient / 4,
                               3 * cxClient / 4, 3 * cyClient / 4,
                                   cxClient / 4,     cyClient / 4) ;

               EndPaint (hwnd, &ps) ;
```

```
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 4-12 LINEDEMO程序

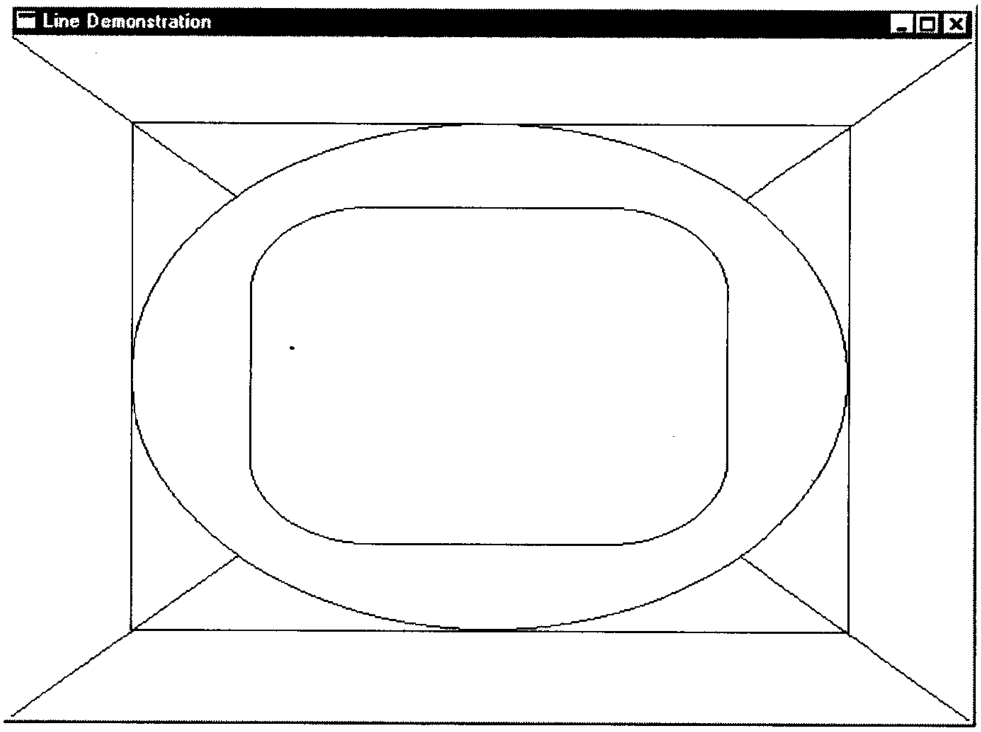


图 4-13 LINEDEMO显示

4.4.2 Bezier 样条

“样条”这个词从前指的是一片木头、橡皮或者金属，用来在纸上画曲线。比如说，如果你有一些分开的图上的点，你想要在它们之间画一条曲线（内插或者外插），你首先将这些点描在绘图纸上，然后，将样条定在这些点上，并用铅笔沿着样条绕着这些点弯曲的方向画曲线（别笑，这听起来好像十九世纪的技术，不过据我个人所知，这些死板的样条就如同计算尺（ slide rule ）一样，在十五年前还在为保险统计师们所使用）。

当然，时至今日，样条已经变成了数学公式，有很多种不同的样条，它们各有千秋。Bezier 样条是计算机程序设计中用得最广的样条之一，它是直到最近才被加到操作系统级的图形支持中的。1960 年 Renault 汽车公司正在进行由手工设计车体（要用到粘土）到计

算机支持的设计的转变。他们需要一些数学工具，而 Pierre Bezier 找到了一套公式，最后表明这套公式对付这样的任务非常有用。

此后，二维的 Bezier 样条成了计算机图形学中最有用的曲线（在直线和椭圆之后）。例如，在 PostScript 中，所有曲线都用 Bezier 曲线表示——椭圆线用 Bezier 来逼近（TrueType 使用一种更简单更快速的样条）。

一条二维的 Bezier 样条由四个点定义——两个端点和两个控制点。曲线的端点在两个端点上，控制点就好像“磁石”一样把曲线从两个端点间的直线处拉走。这一点可以通过叫做“ Bezier ”交互式的程序最好的表明，如图 4-14 所示。

BEZIER.MAK

```
#----------------------
# BEZIER.MAK make file
#----------------------

bezier.exe : bezier.obj
    $(LINKER) $(GUIFLAGS) -OUT:bezier.exe bezier.obj $(GUILIBS)

bezier.obj : bezier.c
    $(CC) $(CFLAGS) bezier.c
```

BEZIER.C

```
/*---------------------------------------
   BEZIER.C -- Bezier Splines Demo
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Bezier" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
```

```
     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Bezier Splines",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DrawBezier (HDC hdc, POINT apt [])
     {
     PolyBezier (hdc, apt, 4) ;

     MoveToEx (hdc, apt[0].x, apt[0].y, NULL) ;
     LineTo   (hdc, apt[1].x, apt[1].y) ;

     MoveToEx (hdc, apt[2].x, apt[2].y, NULL) ;
     LineTo   (hdc, apt[3].x, apt[3].y) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static POINT apt [4] ;
     HDC          hdc ;
     int          cxClient, cyClient ;
     PAINTSTRUCT  ps ;

     switch (iMsg)
          {
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               apt[0].x = cxClient / 4 ;
               apt[0].y = cyClient / 2 ;

               apt[1].x = cxClient / 2 ;
               apt[1].y = cyClient / 4 ;

               apt[2].x =     cxClient / 2 ;
               apt[2].y = 3 * cyClient / 4 ;

               apt[3].x = 3 * cxClient / 4 ;
```

```
          apt[3].y =     cyClient / 2 ;

          return 0 ;

     case WM_MOUSEMOVE :
          if (wParam & MK_LBUTTON || wParam & MK_RBUTTON)
               {
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
               DrawBezier (hdc, apt) ;

               if (wParam & MK_LBUTTON)
                    {
                    apt[1].x = LOWORD (lParam) ;
                    apt[1].y = HIWORD (lParam) ;
                    }

               if (wParam & MK_RBUTTON)
                    {
                    apt[2].x = LOWORD (lParam) ;
                    apt[2].y = HIWORD (lParam) ;
                    }

               SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
               DrawBezier (hdc, apt) ;
               ReleaseDC (hwnd, hdc) ;
               }

          return 0 ;

     case WM_PAINT :
          InvalidateRect (hwnd, NULL, TRUE) ;

          hdc = BeginPaint (hwnd, &ps) ;

          DrawBezier (hdc, apt) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }

return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 4-14 BEZIER程序

由于这个程序要用到一些我们在第六章才讲的鼠标处理，所以，我不在这里讨论它的内部运作（不过，这也是浅显的），而是用这个程序来实验性地操纵 Bezier 样条。在这个程序中，两个顶点设置在客户区的上下居中、左右位于 1/4 和 3/4 处的位置；两个控制点可

以改变，按住鼠标左键或右键并拖动鼠标可以分别改动两个控制点之一。图4-15是一个典型例子。

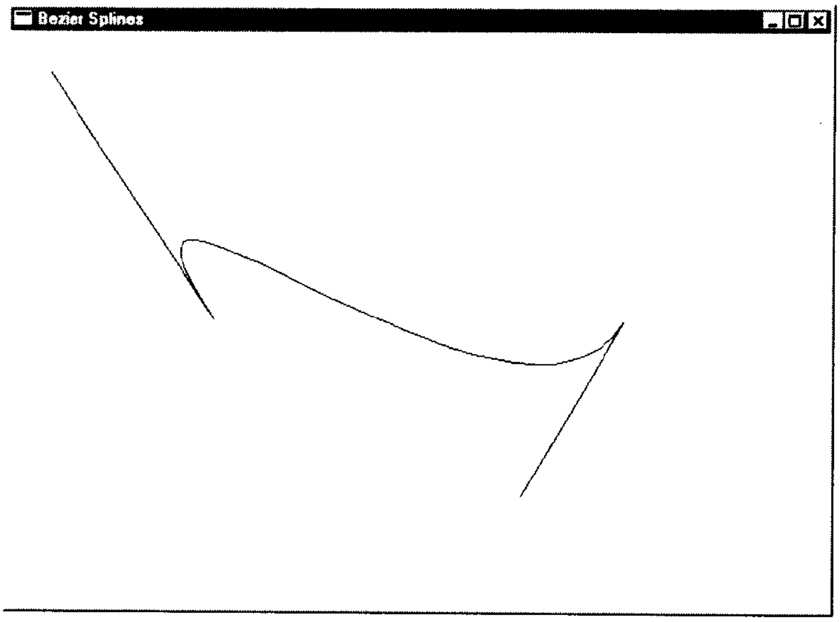


图4-15 BEZIER程序的显示

除了 Bezier 样条自身，程序还从第一个控制点向左边的第一个端点（也叫做开始点）画一条直线，并从第二个控制点向右边的端点画一条直线。

由于下面几个特点， Bezier 样条在计算机辅助设计中非常有用。

首先，经过少量练习，就可以把曲线调整到非常接近想要的形状。

其次， Bezier 样条非常好控制。对于有的样条种类来说，曲线不经过任何一个定义该曲线的点。 Bezier 曲线总是由其两个端点开始和结束的（这是在推导 Besier 公式时所做的假设之一）。另外，有些形式的样条有奇点，在这些点处曲线偏向无穷远，这在基于计算机的设计中通常是很不合适的。事实上， Bezier 曲线总是受限于一个四边形(叫做"凸壳")，这个四边形由端点和控制点连接而成。

第三个特点涉及端点和控制点之间的关系。曲线总是与第一个控制点到起点的直线相切，并保持同一方向；同时，也与第二个控制点到终点的直线相切，并保持同一方向。这是用于推导 Bezier 公式是所做的另外两个假设。

第四, Bezier 样条通常比较具有美感。我知道这是一个主观评价的问题，不过，并非只有我才这样想。

在 Windows 95 之前，你必须利用 Polyline 来自己创建 Bezier 样条，并且还需要知道下面的 Bezier 曲线的参数方程。起点是（ x0, y0 ），终点是（ x3, y3 ），两个控制点是（ x1, y1 ）和（ x2, y2 ），随着 t 的值从 0 到 1 的变化，就可以画出曲线。

```
x(t) = (1-t)3x0 + 3t(1-t)2x1 + 3t2(1-t)x2 + t3x3
y(t) = (1-t)3y0 + 3t(1-t)2y1 + 3t2(1-t)y2 + t3y3
```

在 Windows 95 中，你不需要知道这些公式，要画一条或者多条首尾相接的 Bezier 样

条，就调用：

```
PolyBezier (hdc, pt, iCount) ;
```

或者

```
PolyBezierTo (hdc, pt, iCount) ;
```

两种情况下，pt 都是 POINT 结构的数组。对 PolyBezier，前四个点（按照顺序）给出 Bezier 曲线的起点、第一个控制点、第二个控制点和终点。此后的每一条 Bezier 曲线只要给出三个点，因为后一条 Bezier 曲线的起点就是前一条 Bezier 曲线的终点，如此类推。iCount 参数等于 1 加上你所绘制的这些首尾相接曲线条数的三倍。

PolyBezierTo 函数使用当前点作为第一个起点，第一条以及后续的 Bezier 样条都只需要给出三个点。当函数返回时，当前点被设置为最后一个终点。

一点提示：在画一系列的 Bezier 样条时，只有当第一条 Bezier 曲线的第二个控制点、第一条 Bezier 曲线的终点（也就是第二条曲线的起点）和第二条 Bezier 曲线的第一个控制点线性相关时，也就是说这三个点在同一条直线上时，曲线在连接点处才是光滑的。

4.4.3 使用现有画笔（stock pen）

当你调用这一节中讨论的任何画线函数时，Windows 使用设备环境中当前选中的"画笔"来画线。画笔决定线的色彩、宽度和风格，风格可以是实线、点划线或者虚线，缺省设备环境中画笔为 BLACK_PEN。不管映射方式是什么，这种画笔都画出一个象素宽的黑色实线来。BLACK_PEN 是 Windows 提供的三种现有画笔之一，其它两种是 WHITE_PEN 和 NULL_PEN，NULL_PEN 什么都不画。你也可以自己定制画笔。

Windows 程序使用句柄来引用画笔。Windows 头文件中包含一个叫做 HPEN 的类型定义，即画笔的句柄，可以定义这个类型的变量（例如 hPen）：

```
HPEN hPen ;
```

调用 GetStockObject，可以获得现有画笔的句柄。例如，假设你想使用名为 WHITE_PEN 的现有画笔，可以如下获取画笔的句柄：

```
hPen = GetStockObject (WHITE_PEN) ;
```

接着，必须使画笔成为设备环境中当前被选中的画笔，这需要调用 SelectObject：

```
SelectObject(hdc, hPen) ;
```

在这个调用之后，你画的线将使用 WHITE_PEN，直到你将另外一个画笔选入设备环境中或者释放设备环境为止。

你也可以不显式地定义 hPen 变量，而将 GetStockObject 和 SelectObject 调用合并成一个语句：

```
SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
```

如果想返回到使用 BLACK_PEN 的状态，可以用一条语句获取这种现有画笔的句柄，并将它选入设备环境中：

```
SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
```

SelectObject 的返回值是此调用前设备环境中的画笔句柄。如果启用一个新的设备环境，并调用:

```
hPen = SelectObject (hdc, GetStockObject(WHITE_PEN)) ;
```

则设备环境中的当前画笔将为 WHITE_PEN，变量 hPen 将会是 BLACK_PEN 的句柄。以后，可以通过调用:

```
SelectObject (hdc, hPen) ;
```

重新将 BLACK_PEN 选入设备环境中。

4.4.4 画笔的创建、选择和删除

尽管使用现有画笔非常方便，但却受限于实心的黑画笔、实心的白画笔或者没有画笔这三种情况。如果想得到更丰富多彩的效果，就必须创建自己的画笔。这一过程通常是：使用函数 CreatePen 或者 CreatePenIndirect 创建一个“逻辑画笔”，这仅仅是对画笔的描述；（你也可以使用 ExtCreatePen，我将在这一章后面讨论 GDI 路径时，说明这个函数。）这些函数返回逻辑画笔的句柄；然后，调用 SelectObject 将画笔选入设备环境中。现在，就可以使用新的画笔来画线了。在任何时候，都只能有一种画笔选入设备环境。在释放设备环境（或者在选择了另一种画笔到设备环境中）之后，就可以调用 DeleteObject 来删除所创建的逻辑画笔了。在删除后，该画笔的句柄就不再有效了。

逻辑画笔是一种“ GDI 对象”。你可以创建和使用画笔，但画笔并不属于你的程序，画笔实际上是属于 GDI 模块的。画笔是你可以创建的六种 GDI 对象之一，其它五种是刷子、位图、区域、字体和调色板。除了调色板之外，这些对象都是通过 SelectObject 选入设备环境中的。

在使用画笔等 GDI 对象时，应该遵守以下三条规则:

- 最后要删除自己创建的所有 GDI 对象。
- 当 GDI 对象正在一个有效的设备环境中时，不要删除它。
- 不要删除现有对象。

这些规则当然是不无道理的，不过有时还挺微妙的。下面我们将举些例子来帮助理解这些规则。

CreatePen 函数的语法形如:

```
hPen = CreateObject (iPenStyle, iWidth, rgbColor) ;
```

其中，iPenStyle 参数确定画笔是实线、点线还是虚线，该参数可以是 Windows.头文件中定义的以下标识符，图 4-16 显示了每种风格产生的线类型。

对于 PS_SOLID、PS_NULL 和 PS_INSIDEFRAME 风格，iWidth 参数是画笔的宽度。iWidth 值为 0 则意味着画笔宽度为一个象素。现有画笔是一个象素宽。如果指定的是一种点画线或者虚线式画笔风格，同时又指定一个大于 1 的物理宽度，那么 Windows 将使用实线画笔来代替。

PS_SOLID
PS_DASH
PS_DOT
PS_DASHDOT
PS_DASHDOTDOT
PS_NULL
PS_INSIDEFRAME

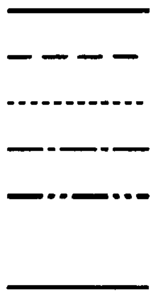


图 4-16 七种画笔风格

CreatePen 的 rgbColor 参数是一个无符号长整数，它指定画笔的色彩。对于除了 PS_INSIDEFRAME 之外的画笔风格，如果将画笔选入设备环境中，Windows 会将 rgbColor 参数转换为设备所表示的最相近的纯色。PS_INSIDEFRAME 风格是唯一一种可以使用一种抖动色的画笔风格，并且只有在宽度大于 1 的情况下才如此。在与定义一个填充区域的函数一起使用时，PS_INSIDEFRAME 风格还有另外一个奇特之处：对于除了 PS_INSIDEFRAME 以外的所有画笔风格来说，如果用来画边界框的画笔宽度大于 1 个象素，那么画笔将被居中对齐在边界框线上，这样边界框线的一部分将位于边界框之外；而对于 PS_INSIDEFRAME 画笔风格来说，整条边界框线都被画在边界框之内。

你也可以通过建立一个类型为 LOGPEN(“逻辑画笔”)的结构，并调用 CreatePenIndirect 来创建一个画笔。如果你的程序使用许多能在源代码中初始化的画笔，那么使用这种方法将有效得多。首先定义一个类型为 LOGPEN 的结构变量，如 logpen:

```
LOGPEN logpen ;
```

这一结构有三个成员：lopnStyle(UINT)是画笔风格，lopnWidth(POINT)是按逻辑单位度量的画笔宽度，lopnCoor(COLORREF)是画笔色彩。lopnWidth 是一个 POINT 结构，不过 Windows 只把 lopnWidth.x 的值作为画笔宽度而忽略 lopnWidth.y 。以 lopen 结构的地址为参数调用 CreatePenIndirect 就可以创建画笔了：

```
hPen = CreatePenIndirect (&lopen) ;
```

你也可以获取已经存在的画笔的逻辑画笔信息。如果已经有了一个画笔的句柄，则可以使用 GetObject 调用将定义逻辑画笔的数据拷贝到一个类型为 LOGPEN 的结构中:

```
GetObject (hPen, sizeof (LOGPEN), (LPVOID) &logpen) ;
```

注意，CreatePen 和 CreatePenIndirect 函数不需要设备环境句柄作为参数。这些函数创建与设备环境没有联系的逻辑画笔。直到调用 SelectObject 之后，画笔才与设备环境发生联系。因此，可以对不同的设备(如屏幕和打印机)使用相同的逻辑画笔。

下面是创建、选择和删除画笔的一种方法。假设你的程序使用三种画笔——一种宽度为 1 的黑画笔、一种宽度为 3 的红画笔和一种黑色点式画笔，你可以先定义三个变量来存放这些画笔的句柄:

```
static HPEN hPen1, hPen2, hPen3;
```

在处理 WM_CREATE 期间，你可以创建这三种画笔:

```
hPen1 = CreatePen (PS_SOLID, 1, 0) ;
hPen2 = CreatePen (PS_SOLID, 3, RGB(255, 0, 0)) ;
hPen3 = CreatePen (PS_DOT, 0, 0) ;
```

在处理 WM_PAINT 期间，或者是在你拥有一个设备环境的有效句柄的任何时间，你都可以将这三个画笔之一选入设备环境中并用它来画线。

```
SelectObject (hdc, hPen2) ;
[画线函数]
SelectObject (hdc, hPen1) ;
[其它画线函数]
```

在处理 WM_DESTROY 期间，你可以删除你创建的三种画笔:

```
DeleteObject (hPen1) ;
DeleteObject (hPen2) ;
DeleteObject (hPen3) ;
```

这是创建、选择和删除画笔的最直接的方法。但是逻辑画笔需要在整个程序运行期间占用存储，为此，你可能想要在每个 WM_PAINT 消息期间创建画笔，并在调用 EndPaint 之后删除它们(你可以在调用 EndPaint 之前删除它们，但是要小心，不要删除设备环境中当前选择的画笔)。

你可能还希望随时创建画笔，并将 CreatePen 和 SelectObject 调用组合到同一个语句中:

```
SelectObject (hdc, CreatePen(PS_DASH, 0, RGB(255, 0, 0))) ;
```

现在再开始画线，你将使用一个红色虚线画笔。在画完红色虚线之后，可以删除画笔。坏了！由于没有保存画笔句柄，怎么才能删除这些画笔呢？不要紧，请记住， SelectObject 将返回设备环境中上一次选择的画笔句柄。所以，你可以通过调用 SelectObject 将 BLACK_PEN 选入设备环境中，并删除从 SelectObject 返回的值:

```
DeleteObject (SelectObject (hdc, GetStockObject(BLACK_PEN))) ;
```

下面是另一种方法，在将新创建的画笔选入设备环境中时，保存 SelectObject 返回的画笔句柄:

```
hPen = SelectObject (hdc , CreatePen (PS_DASH, 0, RGB(255, 0, 0))) ;
```

现在 hPen 是什么呢？如果这是在获取设备环境之后第一次调用 SelectObject ，则 hPen 是 BLACK_PEN 对象的句柄。现在，可以将 hPen 选入设备环境中，并删除所创建的画笔(第二次 SelectObject 调用返回的句柄)，只要一条语句即可：

```
DeleteObject (SelectObject(hdc, hPen)) ;
```

4.4.5 填充空隙

使用点式画笔和虚线画笔会产生一个有趣的问题：点和虚线之间的空隙会怎样呢？空隙的着色取决于背景模式和设备环境中所定义的背景色彩属性。缺省背景模式为 OPAQUE，在这种方式下，Windows 使用背景色来填充空隙，缺省的背景色为白色。这与许多程序在窗口类中用 WHITE_BRUSH 来擦除窗口背景的做法是一致的。

你可以通过如下调用来改变 Windows 用以填充空隙的背景色：

```
SetBkColor (hdc, rgbColor) ;
```

与画笔色彩所使用的 rgbColor 值一样，Windows 将这里的背景色转换为纯色。可以通过调用 GetBkColor 来获取设备环境中定义的当前背景色。

通过将背景模式转换为 TRANSPARENT，就可以阻止 Windows 填充空隙：

```
SetBkMode (hdc, TRANSPARENT) ;
```

此后，Windows 将忽略背景色，并且不填充空隙。可以通过调用 GetBkMode 来获取当前背景模式（TRANSPARENT 或者 OPAQUE)。

4.4.6 绘图方式

设备环境中定义的绘图方式也影响显示器上所画线的外观。设想画这样一条直线，它的色彩由画笔色彩和画线区域原来的色彩共同决定。设想用同一种画笔在白色表面上画出黑线而在黑色表面上画出白线，而且不用知道表面是什么色彩。这样的功能对你有用吗？通过绘图方式的设置，这些都可以实现。

当 Windows 使用画笔来画线时，它实际上执行画笔象素与目标位置处原来象素之间的某种按位布尔运算。象素间的按位布尔运算叫做“光栅运算”，简记为“ROP”。由于画一条直线只涉及两种象素（画笔和目标），因此这种布尔运算又称为“二元光栅运算”，简记为“ROP2”。Windows 定义了 16 种 ROP2 代码，指示 Windows 如何组合画笔象素和目标象素。在缺省设备环境中，绘图方式定义为 R2_COPYPEN，这意味着 Windows 只是将画笔象素拷贝到目标象素，这也是我们通常所熟知的。此外，还有 15 种 ROP2 码。

16 种不同的 ROP2 码是怎样得来的呢？为了示范的需要，我们假设使用单色系统，目标色（窗口客户区的色彩）为黑色（用 0 来表示）或者白色（用 1 来表示），画笔也可以为黑色或者白色。用黑色或者白色画笔在黑色或者白色目标上画图有四种组合：白笔与白目标、白笔与黑目标、黑笔与白目标、黑笔与黑目标。

画笔在目标上绘制后会得到什么呢？一种可能是不管画笔和目标的色彩，画出的线总是黑色的，这种绘图方式由 ROP2 代码 R2_BLACK 指示。另一种可能是只有当画笔与目标都为黑色时，画出的结果才是白色，其它情况下画出的都是黑色。尽管这似乎有些奇怪，

Windows 还是为这种方式起了一个名字，叫做 R2_NOTMERGEPEN 。 Windows 执行目标象素与画笔象素的按位“或”运算，然后翻转所得色彩。

下表显示了所有 16 种绘图方式，表中指示了画笔色彩(P)与目标色彩(D)是如何组合而成结果色彩的。在标有“布尔运算”的那一栏中，用 C 语言的表示法给出了目标象素与画笔象素的组合方式。

画笔(P)：	1	1	0	0	逻辑	
目标(D)：	1	0	1	0	操作	绘图模式
结果：	0	0	0	0	0	R2_BLACK
	0	0	0	1	~(P | D)	R2_NOTMERGEPEN
	0	0	1	0	~P & D	R2_MASKNOTPEN
	0	0	1	1	~P	R2_NOTCOPYPEN
	0	1	0	0	P & ~D	R2_MASKPENNOT
	0	1	0	1	~D	R2_NOT
	0	1	1	0	P ^ D	R2_XORPEN
	0	1	1	1	~(P & D)	R2_NOTMASKPEN
	1	0	0	0	P & D	R2_MASKPEN
	1	0	0	1	~(P ^ D)	R2_NOTXORPEN
	1	0	1	0	D	R2_NOP
	1	0	1	1	~P | D	R2_MERGENOTPEN
	1	1	0	0	P	R2_COPYPEN (缺省)
	1	1	0	1	P | ~D	R2_MERGEPENNOT
	1	1	1	0	P | D	R2_MERGEPEN
	1	1	1	1	1	R2_WHITE

你可以通过调用

```
SetROP2 (hdc, iDrawMode) ;
```

在设备环境中设置某种绘图方式，其中， iDrawMode 是表中“绘图方式”一栏中给出的值之一。你可以用函数：

```
iDrawMode = GetROPS (hdc) ;
```

来获取当前绘图方式。设备环境中的缺省设置为 R2_COPYPEN ，它用画笔色彩替代目标色彩。在 R2_NOTCOPYPEN 方式下，若画笔为黑色，则画成白色；若画笔为白色，则画成黑色。 R2_BLACK 方式下，不管画笔和背景色为何种色彩，总是画成黑色。与此相反， R2_WHITE 方式下总是画成白色。 R2_NOP 方式就是“不操作”，让目标保持不变。

现在，我们来看一个纯单色系统。实际上，在单色显示器上， Windows 可以通过抖动来模拟不同灰度。在抖动生成的背景上使用画笔时， Windows 简单地执行以象素为单位的

位运算。R2_NOT 方式总是翻转目标，不管画笔的色彩是什么。在不知道背景色彩时，这种方式很有用，因为这样能保证画笔的效果总是可见的（当然，只能说是几乎总是可见的，因为如果背景为 50%的灰度，那么效果实际上是不可见的）。我将在第 6 章的 BLOKOUT 程序中演示 R2_NOT 的使用。

4.5 绘制填充区域

现在再迈出一步，从画线到画图形。Windows 中七个用来画带边缘的填充图形的函数列于下表中：

函数	图形
Rectangle	直角矩形
Ellipse	椭圆
RoundRect	圆角矩形
Chord	椭圆周上的弧，两端以弦连接
Pie	椭圆上的饼图
Polygon	多边形
PolyPolygon	多个多边形

Windows 用设备环境中选择的当前画笔来画图形的边界框，边界框还使用当前背景方式、背景色彩和绘图方式，这跟 Windows 画线时一样。关于直线的一切也适用于这些图形的边界框。

图形以当前设备环境中选择的刷子来填充。缺省情况下，使用现有对象 WHITE_BRUSH，这意味着图形内部将画为白色。Windows 定义六种现有刷子：WHITE_BRUSH、LTGRAY_BRUSH、GRAY_BRUSH、DKGRAY_BRUSH、BLACK_BRUSH 和 NULL_BRUSH(也叫 HOLLOW_BRUSH)。

你可以将任何一种现有画刷选入你的设备环境中，就和你选择一种画笔一样。Windows 将 HBRUSH 定义为刷子的句柄，所以可以先定义一个刷子句柄变量:

```
HBRUSH hBrush ;
```

你可以通过调用 GetStockObject 获取 GRAY_BRUSH 的句柄:

```
hBrush = GetStockObject (GRAY_BRUSH) ;
```

你可以使用 SelectObject 将它选入设备环境中:

```
SelectObject (hdc, hBrush) ;
```

现在，如果你要画一个图形，其内部将为灰色。

如果你想画一个没有边界框的图形，可以将 NULL_PEN 选入设备环境中：

```
SelectObject (hdc, GetStockObject(NULL_PEN)) ;
```

如果你想画出图形的边界框，但不填充内部，则将 NULL_BRUSH 选入设备环境中：

```
SelectObject (hdc, GetStockObject(NULL_BRUSH)) ;
```

你也可以定制画刷，就如同你定制画笔一样。我们将马上谈到这个问题。

4.5.1 Polygon 函数和多边形填充方式

我已经讨论过了前五个区域填充函数，Polygon 是第六个画带边界框的填充图形的函数，该函数的调用与 Polyline 函数相似:

```
Polygon (hdc, pt, iCount) ;
```

pt 参数是 POINT 结构的一个数组，iCount 是点的数目。如果该数组中的最后一个点与第一个点不同，则 Windows 将会再加一条线，将最后一个点与第一个点连起来(在 Polyline 函数中，Windows 不会这么做)。

采用两种方式之一，Windows 使用当前画刷来填充这个带边界的区域。至于选用两种方式中的哪一种，则取决于设备环境中定义的当前多边形填充方式。在缺省情况下，多边形填充方式为 ALTERNATE，这意味着 Windows 只填充那些可以通过穿越奇数条线(1，3，5 等等）从多边形的外面到达的内部，其它内部不被填充。也可以将多边形填充方式设置为 WINDING，在这种情况下，Windows 一般填充所有内部区域。使用:

```
SetPolyFillMode (hdc, iMode) ;
```

来设置填充方式。

可以用下面的五角形来演示这两种多边形填充方式。在图 4-17 中，左边的星是用 ALTERNATER 方式画的，右边的星是用 WINDING 方式画的。

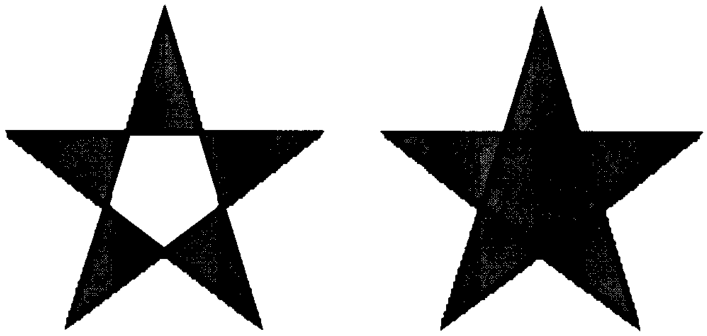


图 4-17 用两种多边形填充方式画出的图：ALTERNATE(左)和WINDING(右)

4.5.2 用画刷填充内部

Rectangle 、 RoundRect 、 Ellipse 、 Chord 、 Pie 、 Polygon 和 PolyPolygon 图形的内部是用选入设备环境中的当前画刷（也称为“模式”）来填充的。画刷是一个 8*8 的位图，它水平和垂直地重复使用来填充内部区域。

当 Windows 用抖动的方法来显示多于可从显示器上得到的色彩时，实际上是将画刷用于色彩。在单色系统上， Windows 能够使用黑色和白色象素的抖动创建 64 种不同的灰色，更精确地说， Windows 能够创建 64 种不同的单色画刷。对于纯黑色， 8*8 位图中的所有位均为 0 。第一种灰色有 1 位为 1 ，第二种灰色有两位为 1 ，以此类推，直到 8*8 位图中所有位均为 1 ，这就是白色。在彩色视频系统上，抖动色也是位图，并且可以得到更多的色彩。

Windows 还有四个函数，可以让你创建逻辑画刷，然后就可使用 SelectObject 将画刷选入设备环境中。与逻辑画笔一样，逻辑画刷也是 GDI 对象。你创建的所有画刷都必须被删除，但是当它还在设备环境中时不能将其删除。

下面是创建逻辑画刷的第一个函数:

```
hBrush = CreateSolidBrush (rgbColor) ;
```

函数中的 Solid 这个词并不是指画刷为纯色。在将画刷选入设备环境中时， Windows 为抖动色创建一个 8*8 位图，并为画刷使用该位图。

你还可以使用由水平、垂直或者倾斜的线组成的“影线标记(hatch marks)”来创建画刷，这种风格的画刷对着色条形图的内部和在绘图仪上进行绘图最有用。创建影线画刷的函数为:

```
hBrush = CreateHatchBrush (iHatchStyle, rgbColor) ;
```

iHatchStyle 参数描述影线标记的外观，此参数可以是 HS_HORIZONTAL 、 HS_VERTICAL 、 HS_FDIAGONAL 、 HS_BDIAGONAL 、 HS_CROSS 中之一，图 4-18 显示了每种风格所产生的影线标记类型。

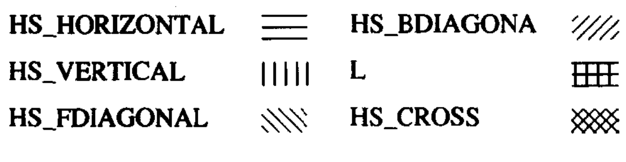


图 4-18 六种影线画刷风格

CreateHatchBrush 中的 rgbColor 参数是影线的色彩。在将画刷选入设备环境中时，Windows 将这种色彩转换为与之最相近的纯色。影线之间的区域根据设备环境中定义的背景方式和背景色来着色。如果背景方式为 OPAQUE ，则用背景色（它也被转换为纯色）来填充线之间的空间。在这种情况下，影线和填充色都不能是抖动色。如果背景方式为

TRANSPARENT ，则 Windows 只画出影线，不填充它们之间的区域。由于画刷总是 8*8 的位图，所以因显示设备分辨率的不同，影线画刷的外观也会发生变化。图 4-18 所示的每种影线标记都是画在一个 32*16 象素的矩形中的，这意味着 8*8 位图在水平方向上重复 4 次，在垂直方向上重复 2 次。在一个 300dpi 的激光打印机上，同样的 32*16 象素矩阵将占据大约 1/9 英寸宽， 1/19 英寸高的区域。

你也可以使用 CreatePatternBrush 来创建自己的基于位图的画刷:

```
hBrush = CreatePatternBrush (hBitmap) ;
```

我将在这一章后面讨论位图时再来讨论这个函数。

Windows 还提供一个函数，它集成了其它三个创建画刷的函数(CreateSolidBrush 、 CreateHatchBrush 、 CreatePatternBrush):

```
hBrush = CreateBrushIndirect (&logbrush) ;
```

变量 logbrush 是一个类型为 LOGBRUSH(“逻辑画刷”)的结构，该结构的三个域如下表所示， lbStyle 域的值确定了 Windows 如何解释其它两个域的值。

lbStyle(UINT)	lbColor(COLORREF)	lbHatch(LONG)
BS_SOLID	画刷的色彩	忽略
BS_HOLLOW	忽略	忽略
BS_HATCHED	影线的色彩	影线画刷风格
BS_PATTERN	忽略	位图的句柄

前面我们用 SelectObject 将逻辑画笔选入到设备环境中，用 DeleteObject 删除画笔，用 GetObject 来获取一个逻辑画笔的信息。对于画刷，同样能使用这三个函数。一旦你获取到了画刷句柄，就可以使用 SelectObject 将该画刷选入设备环境中:

```
SelectObject (hdc, hBrush) ;
```

然后，你可以使用 DeleteObject 函数删除所创建的画刷:

```
DeleteObject (hBrush) ;
```

但是，不要删除当前被选入设备环境中的画刷。如果你需要获取画刷的信息，可以调用 GetObject:

```
GetObject (hBrush , sizeof (LOGBRUSH), (LPVOID)&logbrush) ;
```

其中， logbrush 是一个类型为 LOGBRUSH 的结构。

4.6 映 射 方 式

到目前为止，我都是假设我们是相对于客户区左上角，以象素为单位进行绘图的。这是缺省状况，但不是你的唯一选择。

事实上，“映射方式”是一种影响几乎任何客户区绘图的设备环境属性。另外有四种设备环境属性——窗口原点、视口原点、窗口范围和视口范围——与映射方式密切相关。

大多数 GDI 绘图函数需要坐标值或大小。例如，下面是 TextOut 函数:

```
TextOut(hdc, x, y, szBuffer, iLength) ;
```

参数 x 和 y 分别指示文本开始位置的横坐标和纵坐标。参数 x 是在水平轴上的位置，参数 y 是在垂直轴上的位置，通常用(x， y)来表示这个点。

在 TextOut 中，以及在几乎所有 GDI 函数中，这些坐标值使用的都是一种“逻辑单位”。Windows 必须将逻辑单位转换为“设备单位”，即象素。这种转换是由映射方式、窗口和视口的原点以及窗口和视口的范围所控制的。映射方式还隐含地给出了 x 轴和 y 轴的指向(orientation)；也就是说，它确定了当你在向显示器的左或者右移动时 x 的值是增大还是减小，以及在上下移动时 y 的值是增大还是减小。

Windows 定义了 8 种映射方式，它们在 Windows 头文件中相应的标识符和含义列表如下:

		增加值	
映射方式	**逻辑单位**	X－值	Y－值
MM_TEXT	象素	右	下
MM_LOMETRIC	0.1mm	右	上
MM_HIMETRIC	0.01mm	右	上
MM_LOENGLISH	0.01 英寸	右	上
MM_HIENGLISH	0.001 英寸	右	上
MM_TWIPS*	1/1440 英寸	右	上
MM_ISOTROPIC	任意(x=y)	可选	可选
MM_ANISOTROPIC	任意(x!=y)	可选	可选

*Twip 是杜撰的字，意思是“二十分之一点”。点是字型的计量单位，约等于 1/72 英寸，但是在图形系统（如 GDI ）中经常假定为正好 1/72 英寸。一twip 等于 1/20 点，也就是 1/1440 英寸。

你可以使用下面的语句来设置映射方式:

```
SetMapMode (hdc, iMapMode) ;
```

其中， iMapMode 是 8 个映射方式标识符之一。你可以通过调用 GetMapMode 函数获取当前的映射方式:

```
GetMapMode (hdc) ;
```

缺省映射方式为 MM_TEXT 。在这种映射方式下，逻辑单位与物理单位相同，这样我们可以直接以象素为单位进行操作。在一个形如

```
TextOut (hdc, 8, 16, szBuffer, iLength) ;
```

的 TextOut 调用中，文本从距离客户区左端 8 象素、距离上端 16 象素的位置处开始。

如果映射方式被设置为 MM_LOENGLISH ，则逻辑单位是百分之一英寸:

```
SetMapMode (hdc, MM_LOENGLISH) ;
```

现在， TextOut 函数调用形如:

```
TextOut (hdc, 50, -100, szBuffer, iLength) ;
```

文本从距离客户区左端 0.5 英寸，距离上端 1 英寸的位置处开始。至于 y 坐标前面的负号，随着我们对映射方式更详细的讨论，将会变得逐渐清楚。其它映射方式允许程序按照毫米、打印机的点大小或者任意单位的坐标轴来指定坐标。

如果你认为使用象素进行工作很合适，那么就不要使用缺省的 MM_TEXT 方式外的任何映射方式。如果需要以实际的英寸或者毫米尺寸显示图象，那么可以从 GetDeviceCaps 中获取所需要的信息，自己再进行缩放。其它映射方式都是避免你自己进行缩放的一个方便途径。

不管在什么映射方式下，在 Windows 函数中指定的所有坐标都必须是有符号整数，范围从 32767 到－32768 。一些使用坐标表示矩形的开始点和结束点的 Windows 函数也要求矩形的宽和高小于或者等于 32767 。

4.6.1 设备坐标和逻辑坐标

你也许会问：如果使用 MM_LOENGLISH 映射方式，是不是将会得到以百分之一英寸为单位的 WM_SIZE 消息呢？绝对不会。 Windows 对所有消息（如 WM_MOVE 、WM_SIZE 和 WM_MOUSEMOVE ），对所有非 GDI 函数，甚至对一些 GDI 函数，永远使用设备坐标。可以这样来考虑：由于映射方式是一种设备环境属性，所以，只有对需要设备环境句柄做参数的 GDI 函数，映射方式才会起作用。 GetSystermMetrics 不是 GDI 函数，所以它总是以设备单位（即象素）为量度来返回大小的。尽管 GetDeviceCaps 是 GDI 函数，需要一个设备环境句柄作为参数，但是 Windows 仍然对 HORZRES 和 VERTRES 以设备单位作为返回值，因为该函数的目的之一就是给程序提供以象素为单位的设备大小。

不过，从 GetTextMetrics 调用中返回的 TEXTMETRICSK 结构的值是使用逻辑单位的。如果在进行此调用时映射方式为 MM_LOENGLISH ， GetTextMetrics 将以百分之一英寸为单位提供字符的宽度和高度。在调用 GetTextMetrics 以获取关于字符的宽度和高度信息时，映射方式必须设置成根据这些信息输出文本时所使用的映射方式。在介绍各种 GDI 函数时，我将同时说明它们是使用设备坐标还是使用逻辑坐标。除了带风格线条中点或短线之间的间距，以及图案中条纹的间距外，迄今为止所讨论的所有函数使用的都是逻辑坐标。

4.6.2 设备坐标系

Windows 将 GDI 函数中指定的逻辑坐标映射为设备坐标。在讨论以各种不同的映射方

式使用逻辑坐标系之前，我们先来看一下 Windows 为视频显示器区域定义的不同的设备坐标系。尽管我们大多数时间在窗口的客户区内工作，但 Windows 在不同的时间使用另外两种设备坐标区域。所有设备坐标系都以象素为单位，水平轴（即 x 轴）上的值从左到右递增，垂直轴（即 y 轴）上的值从上到下递增。

当我们使用整个屏幕时，就根据“屏幕坐标”进行操作。屏幕的左上角为(0, 0)点，屏幕坐标用在 WM_MOVE 消息（对于非子窗口）以及下列 Windows 函数中:CreateWindow 和 MoveWindow（都是对于非子窗口）、GetMessagePos 、GetCursorPos 、SetCursorPos 、GetWindowRect 、WindowFromPoint 以及 SetBrushOrgEx 。它们或者是与窗口无关的函数（如两个光标函数），或者是必须相对于某个屏幕点来移动（或者寻找）窗口的函数。如果以 DISPLAY 为参数调用 CreateDC，以获取整个屏幕的设备环境，则 GDI 调用中指定的逻辑坐标将被映射为屏幕坐标。

“整窗口坐标”以程序的整个窗口为基准，标题条、菜单、滚动条和窗口框都包括在内。而对于普通窗口，点(0, 0)是缩放边框的左上角。整窗口坐标在 Windows 中极少使用，但是如果用 GetWindowDC 获取设备环境，GDI 函数中的逻辑坐标就会转换为客户区坐标。

用函数 ClientToScreen 和 ScreenToClient 可以将客户区坐标转换为屏幕坐标，或者反过来，将屏幕坐标转换为客户区坐标。也可以使用 GetWindowRect 函数获取屏幕坐标下的整个窗口的位置和大小。这三个函数为一种设备坐标转换为另一种提供了足够的信息。

4.6.3 视口和窗口

映射方式定义了 Windows 如何将 GDI 函数中指定的逻辑坐标映射为设备坐标，这里的设备坐标系取决于你用哪个函数来获取设备环境。要继续讨论映射方式，我们需要一些术语：映射方式用于定义从“窗口”(逻辑坐标)到“视口”(设备坐标)的映射。

“窗口”和“视口”这两个词用得并不恰当。在其它图形界面语言中，视口通常包含有剪裁区域的意思，并且，我们已经用窗口来指程序在屏幕上占据的区域。在这里的讨论中，我们必须把关于这些词的先入之见抛到一边。

“视口”是基于设备坐标（象素）的。通常，视口和客户区相同，但是，如果你已经用 GetWindowDC 或者 CreateWindowDC 获取了一个设备环境，则视口也可以是指整窗口坐标或者屏幕坐标。点(0, 0)是客户区（或者整个窗口或屏幕）的左上角，x 的值向右增加，y 的值向下增加。

“窗口”是基于逻辑坐标的，逻辑坐标可以是象素、毫米、英寸或者你想要的任何其它单位。GDI 函数使用逻辑窗口坐标。

对于所有的映射方式，Windows 都用下面两个公式来将窗口（逻辑）坐标转化为视口（设备）坐标:

$$xViewport = (xWindow - xWinOrg) * \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) * \frac{yViewExt}{yWinExt} + yViewOrg$$

其中，(xWindow, yWindow)是待转换的逻辑点，(xViewport, yViewport)是转换后的设备坐标点。如果设备坐标是客户区坐标或者整窗口坐标，则 Windows 在画一个对象之前，还必须将这些坐标转换为屏幕坐标。

这两个公式使用了分别指定窗口和视口“原点”的点:(xWinOrg, yWinOrg)是逻辑坐标的窗口原点；(xViewOrg, yViewOrg)是设备坐标的视口原点。在缺省的设备环境中，这两个点均被设置为(0, 0)，但是它们可以被改变。此公式意味着，逻辑点(xWinOrg, yWinOrg)总被映射为设备点(xViewOrg, yViewOrg)。

此公式还使用了两点来指定“范围”：(xWinExt, yWinExt)是逻辑坐标的窗口范围；(xViewExt, yViewExt)是设备坐标的窗口范围。在多数映射方式中，范围是映射方式所隐含的，不能够改变。每个范围自身没有什么意义，但是视口范围与窗口范围的比是逻辑单位转换为设备单位的换算因子。范围可以为负，也就是说，逻辑 x 轴上的值不一定非得在向右时增加；逻辑 y 轴上的值也不一定非得在向下时增加。

Windows 也能将视口（设备）坐标转换为窗口（逻辑）坐标:

$$xWindow = (xViewport - xViewOrg) * \frac{xWinExt}{xViewExt} + xWinOrg$$

$$yWindow = (yViewport - yViewOrg) * \frac{yWinExt}{yViewExt} + yWinOrg$$

Windows 提供了两个函数来让你将设备点转换为逻辑点以及将逻辑点转换为设备点。下面的函数将设备点转换为逻辑点:

```
DPtoLP (hdc, pPoints, iNumber) ;
```

其中，pPoints 是一个指向 POINTS 结构数组的指针，而 iNumber 是要转换的点的个数。你会发现这个函数对于将 GetClientRect(它总是使用设备单位)获取的客户区大小转换为逻辑坐标很有用:

```
GetClientRect (hwnd, &rect) ;
DPtoLP (hdc, (PPOINT) &rect, 2) ;
```

下面的函数将逻辑点转换为设备点:

```
LPtoDP (hdc, pPoints, iNumber) ;
```

4.6.4 MM_TEXT 映射方式

在 MM_TEXT 映射方式下，缺省原点和范围如下所示:

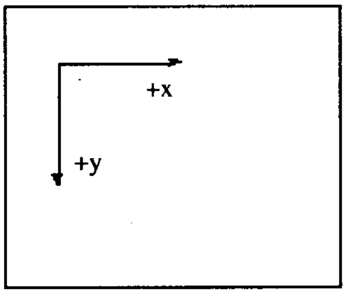
窗口原点: (0， 0) 可以改变

视口原点:	(0， 0)	可以改变
窗口范围:	(1， 1)	不可改变
视口范围:	(1， 1)	不可改变

视口范围与窗口范围的比例为 1，所以不用在逻辑坐标与设备坐标之间进行缩放。上面所给出的公式可以简化为:

```
xViewport = xWindow-xWinOrg+xViewOrg
yViewport = yWindow-yWinOrg+yViewOrg
```

这种映射方式被称为“文本”映射方式，不是因为它对于文本最适合，而是由于轴的方向。我们读文本是从左至右，从上至下的，而 MM_TEXT 以同样的方向定义轴上值的增长方向:



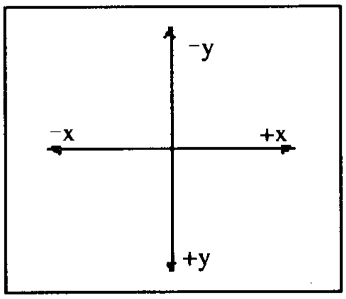
Windows 提供了函数 SetViewPortOrg 和 SetWindowOrg，用来改变视口和窗口的原点，这些函数都具有改变轴的效果，以致(0,0)不再指左上角。一般来说，你会使用 SetViewPortOrg 和 SetWindowOrg 之一，但不会同时使用二者。

我们来看一看这些函数有何效果:如果将视口原点改变为(xViewOrg， yViewOrg)，则逻辑点(0， 0)就会映射为设备点(xViewOrg， yViewOrg)。如果将窗口原点改变为(xWinOrg， yWinOrg)，则逻辑点(xWinOrg, yWinOrg)将会映射为设备点(0， 0)，即左上角。不管对窗口和视口原点作什么改变，设备点(0， 0)始终是客户区的左上角。

例如，假设客户区为 cxClient 个象素宽和 cyClient 个象素高。如果想将逻辑点(0， 0)定义为客户区的中心，可进行如下调用：

```
SetViewportOrgEx (hdc, cxClient/2, cyClient/2, NULL) ;
```

SetViewportOrgEx 的参数总是使用设备单位。现在，逻辑点（0， 0)将被映射为设备点(cxClient/2, cyClient/2)，而客户区的坐标系变成如下形状:



逻辑 x 轴的范围从-cxClient/2 到+cxClient/2，逻辑 y 轴的范围从-cyClient/2 到

+cyClient/2，客户区的右下角为逻辑点(cxClient/2， cyClient/2)。如果你想要从客户区的左上角开始显示文本，则需要使用负坐标:

```
TextOut (hdc, -cxClient/2, -cyClient/2, "Hello", 5) ;
```

用下面的 SetWindowOrgEx 语句可以获得与上面使用 SetViewportOrgEx 同样的效果:

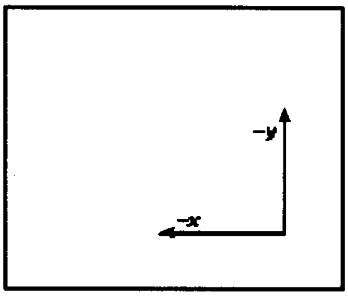
```
SetWindowOrgEx(hdc, -cxClient/2, -cyClient/2, NULL) ;
```

SetWindowOrgEx 的参数总是使用逻辑单位。在这个调用之后，逻辑点 (-cxClient/2, -cyClient/2)被映射为设备点(0， 0)，即客户区的左上角。

你不会将这两个函数一起用，除非你知道这么做的结果:

```
SetViewportOrgEx (hdc, cxClient/2, cyClient/2, NULL) ;
SetWindowOrgEx (hdc, -cxClient/2, -cyClient/2, NULL) ;
```

这意味着逻辑点(-cxClient/2, -cyClient/2)将被映射为设备点(cxClient/2, cyClient/2)，结果是如下所示的坐标系:



你可以使用下面两个函数获取当前视口和窗口的原点:

```
GetViewportOrgEx (hdc, &pt) ;
GetWindowOrgEx (hdc, &pt) ;
```

其中的 pt 是一个 POINT 结构。

由 GetViewportOrgEx 返回的值是设备坐标；而由 GetWindowOrgEx 返回的值是逻辑坐标。

你可能想改变视口或者窗口的原点，以改变窗口客户区内的显示输出——例如，响应用户在滚动条内的输入。但是，改变视口和窗口原点并不能立即改变显示输出，而必须在改变原点之后刷新输出。例如，在第二章的 SYSMETS2 程序中，我们使用了 iVscrollPos 值(垂直滚动条的当前位置)来调整显示输出的 y 坐标:

```
case WM_PAINT :
     BeginPaint (hwnd, &ps) ;

     for (i = 0 ; i < NUMLINES ; i++)
          {
          y = cyChar * (1 - iVscrollPos + i) ;
          [display text]
          }

     EndPaint (hwnd, &ps) ;
```

```
     return 0 ;
```

我们可以使用 SetWindowOrgEx 获得同样的结果:

```
case WM_PAINT :
     BeginPaint (hwnd, &ps) ;

     SetWindowOrgEx (ps.hdc, 0, cyChar * iVscrollPos) ;

     for (i = 0 ; i < NUMLINES ; i++)
          {
          y = cyChar * (1 + i) ;
          [display text]
          }

     EndPaint (hwnd, &ps) ;
     return 0 ;
```

现在， TextOut 函数的 y 坐标的计算不需要 iVscrollPos 的值。这意味着你可以将文本输出函数放到一个例程中，不用将 iVscrollPos 值传给该例程，因为我们是通过改变窗口原点来调整文本显示的。

如果你有使用直角坐标系(即笛卡尔坐标系)的经验，那么将逻辑点(0 ， 0)移到客户区的中央(象我们上面所说的那样)的确值得考虑。但是，对于 MM_TEXT 映射方式来说，还存在着一个小小的问题：笛卡儿坐标系中， y 的值是随着上移而增加的，而 MM_TEXT 定义为下移时 y 的值增加。从这一点来看， MM_TEXT 有点古怪，而下面这五种映射方式都使用通常的增值方法。

4.6.5 “度量”映射方式

Windows 包含五种以物理尺寸来表示逻辑坐标的映射方式。由于 x 轴和 y 轴的逻辑坐标映射为相同的物理单位，这些映射方式能使你画出不变形的圆和矩形。

这五种“度量”映射方式在下表中列出，按照从低精度到高精度的顺序排列。右边的两列分别给出了以英寸和毫米为单位时逻辑单位的大小，以便比较。

映射方式	逻辑单位	英寸	毫米
MM_LOENGLISH	0.01 in.	0.01	0.254
MM_LOMETRIC	0.1 mm	0.00394	0.1
MM_HIENGLISH	0.001 in.	0.001	0.0254
MM_TWIPS*	1/1440 in.	0.000694	0.0176
MM_HIMETRIC	0.01 mm	0.000394	0.01

* 一twip 等于 1/20 点，而一点等于 1/72 英寸，因此一twip 等于 1/1440 英寸。

为了使读者对 MM_TEXT 方式与这些分辨率的对应关系有一个概念，给出以下信息：在一个标准 VGA 显示器上，每个象素宽和高均为 0.325mm ，所以在任何尺度的映射方式下 VGA 设备坐标都较大。

在一个每寸 300 点的激光打印机上，每个象素为 0.0033 英寸，这个分辨率比 MM_LOENGLISH 和 MM_LOMETRIC 高，但是比 MM_HIENGLISH 、 MM_TWIPS 和 MM_HIMETRIC 都低。

缺省原点和范围如下所示:

窗口原点:	(0， 0)	可以改变
视口原点:	(0， 0)	可以改变
窗口范围:	(?， ?)	不能改变
视口范围:	(?， ?)	不能改变

窗口和视口的范围取决于映射方式和设备的纵横比。前面已经提到过，这些范围本身并不重要，但是表示比例时就必须知道。下面是窗口坐标到视口坐标的转换公式:

$$xViewport = (xWindow - xWinOrg) * \frac{xViewExt}{xWinExt} + xViewOrg$$

$$yViewport = (yWindow - yWinOrg) * \frac{yViewExt}{yWinExt} + yViewOrg$$

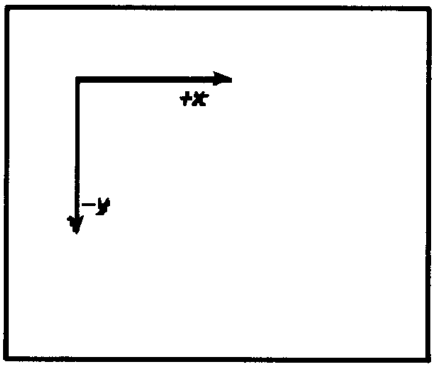
例如，对于 MM_LOENGLISH, Windows 计算的范围如下：

$$\frac{xViewExt}{xWinExt} = \text{每0.01英寸中水平象素数}$$

$$\frac{-yViewExt}{yWinExt} = \text{每0.01英寸中垂直象素数的相反数}$$

对于许多显示设备(如 VGA)，这一比例小于 1 。由于 Windows 完全使用整数进行操作，使用比例而不是一个绝对的换算因子是有必要的，这样可以在逻辑坐标与设备坐标间转换时减少精度的丢失。

请注意垂直轴比例前面的负号，这个负号改变了 y 轴的方向。对于这五种映射方式，y 轴的值随上升而增加。缺省窗口和视口原点均为(0 ， 0)，这一事实有一个有趣的结果。一开始改变为五种映射方式之一时，坐标系如下:



要想在客户区显示任何东西，必须使用负的 y 值。例如下面的代码：

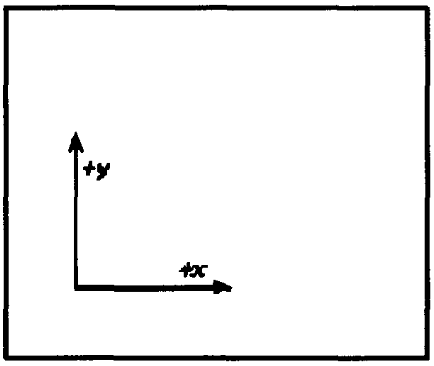
```
SetMapMode (hdc, MM_LOENGLISH) ;
TextOut (hdc, 100, -100, "Hello", 5) ;
```

将把 Hello 显示在距离客户区左边和上边各一英寸的地方。

为了使自己保持头脑清醒，你可能想避免这样做。一种解决办法是将逻辑的(0, 0)点设为客户区的右下角，你可以通过调用 SetViewportOrgEx 来完成：

```
SetViewportOrgEx (hdc, 0, cyClient, NULL) ;
```

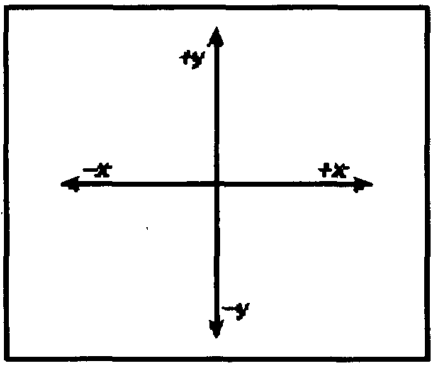
此时的坐标系如下：



另一种方法是将逻辑原点设为客户区的中心：

```
SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
```

此时的坐标系如下：



现在我们有了一个真正的 4 象限笛卡尔坐标系，在 x 轴和 y 轴上有相等的按英寸、毫米或 twip 计算的逻辑单位。

你还可以使用 SetWindowOrgEx 函数来改变逻辑(0， 0)点，但是这稍微困难一些，因为 SetWindowOrgEx 的参数必须使用逻辑单位，先要将(cxClient， cyClient)用 DPtoLP 函数转换为逻辑坐标。假设变量 pt 是类型为 POINT 的结构，下面的代码将逻辑(0， 0)点改变到客户区的中央：

```
pt.x = cxClient ;
pt.y = cyClient ;
DPtoLP (hdc , &pt , 1) ;
SetWindowOrgEx (hdc , -pt.x/2 , -pt.y/2 , NULL) ;
```

4.6.6 "自作主张的"映射方式

剩下的两种映射方式为 MM_ISOTROPIC 和 MM_ANISOPTROPIC 。只有这两种映射方式允许你改变视口和窗口范围，也就是说可以改变 Windows 用来转换逻辑和设备坐标的换算因子。"各向同性(isotropic)"的意思是"在所有方向上都相等"；"各向异性(anisotropic)"的意思相反——"不相等"。与上面所讨论的尺度映射方式相似， MM_ISOTROPIC 使用相同尺度的轴， x 轴上的逻辑单位与 y 轴上的逻辑单位的物理尺寸相等，这对你创建纵横比与显示设备的纵横比无关的图象是有帮助的。

MM_ISOTROPIC 与尺度映射方式之间的区别是，使用 MM_ISOTROPIC ，你可以控制逻辑单位的物理尺寸。如果愿意，你可以根据客户区的大小来调整逻辑单位的物理尺寸，从而使所画的图象总是包含在客户区内，并相应地放大或缩小。例如，第 7 章里的 ANACLOCK("模拟时钟")程序就是"各向同性"图形的一个例子。这个钟总是圆的，在改变窗口大小时，这个图象也随着变化。 Windows 程序完全可以通过调整窗口和视口范围来处理图象大小的变化，因此，不管窗口尺寸怎样变，程序都可以在绘图函数中使用相同的逻辑单位。

有时候 MM_TEXT 和"尺度"映射方式被称为"全约束"映射方式，这就是说，你不能改变窗口和视口的范围以及 Windows 将逻辑坐标换算为设备坐标的方法。 MM_ISOTROPIC 是一个"半约束"的映射方式， Windows 允许你改变窗口和视口范围，但只是调整它们，以便 x 和 y 逻辑单位代表同样的物理尺寸。 MM_ANISOTROPIC 映射方式是"无约束"的，你可以改变窗口和视口范围，但是 Windows 不调整这些值。

4.6.6.1 MM_ISOTROPIC映射方式

如果要想在使用任意的轴时都保证两个轴上的逻辑单位相同，那么 MM_ISOTROPIC 就是理想的映射方式。这时，具有相同逻辑宽度和高度的矩形显示为正方形，具有相同逻辑宽度和高度的椭圆显示为圆。

当你刚开始将映射方式设置为 MM_ISOTROPIC 时， Windows 使用与 MM_LOMETRIC 同样的窗口和视口范围(但是，不要对此有所依赖)。区别在于，你现在可以调用 SetWindowExt 和 SetViewportExtEx 来根据自己的偏好改变范围了，然后， Windows 将调整范围的值，以使两条轴上的逻辑单位有相同的物理距离。

一般说来，你可以用所期望的逻辑窗口的逻辑尺寸来做 SetWindowExtEx 的参数，用客户区的实际宽和高来作 SetViewportExtEx 的参数。 Windows 在调整这些范围时，必须让逻辑窗口适应物理视口，有可能导致客户区的一段落在逻辑窗口的外面。必须在 SetViewportExtEx 调用之前调用 SetWindowExtEx ，以便最有效地使用客户区中的空间。

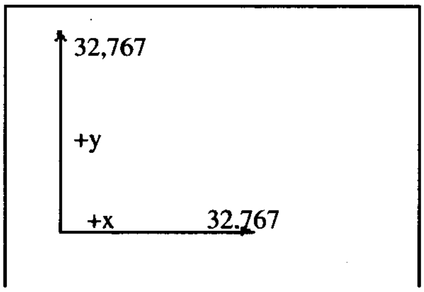
例如，假设你想要一个"传统的"单象限虚拟坐标系，其中(0 ， 0)在客户区的左下角，

宽度和高度的范围都是从 0 到 32767 ，并且希望 x 和 y 轴的单位具有同样的物理尺寸。以下就是所需的程序:

```
SetMapMode (hdc , MM_ISOTROPIC) ;
SetWindowExtEx (hdc , 32767 , 32767 , NULL) ;
SetViewportExtEx (hdc , cxClient , -cyClient , NULL) ;
SetViewportOrgEx (hdc , 0 , cyClient , NULL) ;
```

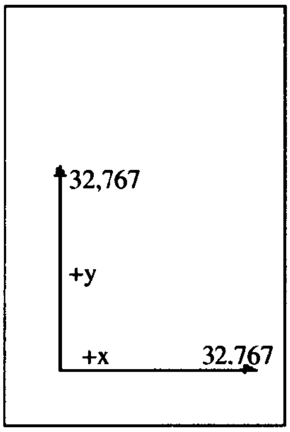
如果其后用 GetWindowExtEx 和 GetViewportExtEx 函数获得了窗口和视口的范围，可以发现，它们并不是先前指定的值。 Windows 将根据显示设备的纵横比来调整范围，以便两条轴上的逻辑单位表示相同的物理尺寸。

如果客户区的宽度大于高度（以物理尺寸为准）， Windows 将调整 x 的范围，以便逻辑窗口比客户区视口窄。这样，逻辑窗口将被放置在客户区的左边:



你不能在客户区的右边超越 x 轴的范围之外显示任何东西，因为这需要一个大于 32767 的逻辑 x 坐标。

如果客户区的高度大于宽度（以物理尺寸为准），那么 Windows 将调整 y 的范围。这样，逻辑窗口将被放置在客户区的下边:



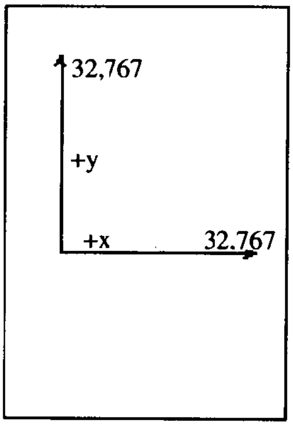
现在，你不能在客户区的顶部显示任何东西，因为这需要一个大于 32767 的逻辑 y 坐标。

如果你希望逻辑窗口总是放在客户区的左上部，那么将前面给出的代码改为:

```
SetMapMode (hdc , MM_ISOTROPIC) ;
```

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 32767, 32767, NULL) ;
SetViewportExtEx (hdc, cxClient, -cyClient, NULL) ;
SetWindowOrgEx (hdc, 0, 32767, NULL) ;
```

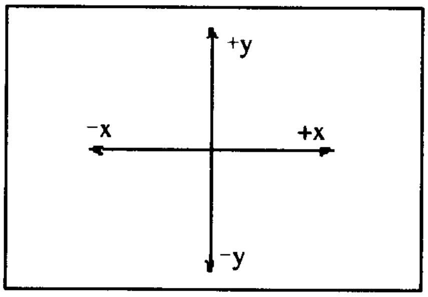
在调用 SetWindowOrgEx 中，我们要求将逻辑点(0, 32767)映射为设备点(0， 0)。现在，如果客户区的高大于宽，则坐标系将被安排为:



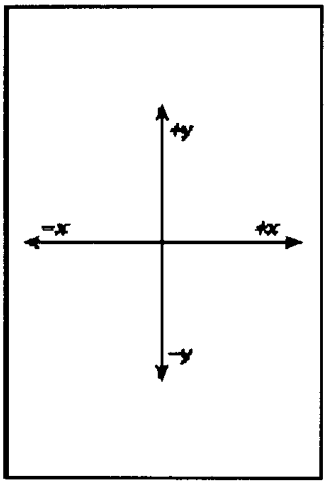
对于一个像 ANACLOCK 那样的图象，你也许想要用一个四象限的笛卡尔坐标系，四个方向的坐标尺度可以任意指定，(0, 0)必须居于客户区的中央。如果你想要每条轴的范围从 0 到 1000，则可以使用以下代码:

```
SetMapMode (hdc, MM_ISOTROPIC) ;
SetWindowExtEx (hdc, 1000, 1000, NULL) ;
SetViewportExtEx (hdc, cxClient/2, -cyClient/2, NULL) ;
SetViewportOrgEx (hdc, cxCleint/2, cyClient/2, NULL) ;
```

如果客户区的宽度大于高度，则逻辑坐标系形如:



如果客户区的高度大于宽度，那么逻辑坐标也会居中:



记住，窗口或者视口范围并不意味着要进行剪裁。在调用 GDI 函数时，你仍然可以随便地使用小于-1000 和大于 1000 的 x 和 y 值。根据客户区的外形，这些点可能看得见，也可能看不见。

在 MM_ISOTROPIC 映射方式下，可以使逻辑单位大于象素。例如，假设你想要一种映射方式，使点(0, 0)在显示屏幕的左上角， y 的值向下增长(和 MM_TEXT 相似)，但是逻辑坐标单位为 1/16 英寸。这种映射方式将允许你从客户区的左上边开始，画一个以 1/16 英寸为单位的标尺：

```
SetMapMode (hdc, MM_ISOTROPIC) ;

SetWindowExtEx (hdc, 160 * GetDeviceCaps (hdc, HORZSIZE) / 254,
                     160 * GetDeviceCaps (hdc, VERTSIZE) / 254, NULL) ;

SetViewportExtEx (hdc, GetDeviceCaps(hdc, HORZRES),
                       GetDeviceCaps (hdc, VERTRES), NULL) ;
```

在这个代码中，视口范围设置为按象素计算的整个屏幕的大小，窗口范围则必须设置为以 1/16 英寸为单位的整个屏幕的大小。 GetDeviceCaps 以 HORZSIZE 为参数，返回以毫米为单位的设备尺寸。如果我们使用浮点数，将把毫米数除以 25.4，转换为英寸，然后，再乘以 16 以转换为 1/16 英寸。但是，由于我们使用的是整数，所以先乘以 160，再除以 254。

对于大多数输出设备，这段代码都会使逻辑单位大于物理单位。在设备上输出的所有东西都将映射为按 1/16 英寸增量的坐标值。当然，这样就不能画两条间隔 1/32 英寸的水平直线，因为这样将需要小数逻辑坐标。

4.6.6.2 MM_ANISOTROPIC：根据需要放缩图象

在 MM_ISOTROPIC 映射方式下设置窗口和视口范围时， Windows 会调整范围，以便两条轴上的逻辑单位具有相同的物理尺度。在 MM_ANISOTROPIC 映射方式下，Windows 不对你所设置的值进行调整，这就是说， MM_ANISOTROPIC 不需要维持正确的纵横比。

使用 MM_ANISOTROPIC 的一种方法是对客户区使用任意坐标，就像我们对

MM_ISOTROPIC 所做的一样。下面的代码将点（0， 0）设置为客户区的左下角， x 轴和 y 轴都从 0 到 32767：

```
SetMapMode (hdc , MM_ANISOTROPIC) ;
SetWindowExtEx (hdc , 32767 , 32767 , NULL) ;
SetViewportExtEx (hdc , cxClient , -cyClient , NULL) ;
SetViewportOrgEx (hdc , 0 , cyClient , NULL) ;
```

在 MM_ISOTROPIC 方式下，相似的代码导致客户区的一部分在轴的范围之外。但是对于 MM_ANISOTROPIC ，客户区的右上角总是（ 32767 ， 32767 ），而不论其尺度为多大。如果客户区不是正方形的 ，则逻辑 x 和 y 的单位具有不同的物理尺度。

前一节在 MM_ISOTROPIC 映射方式下，我们讨论了在客户区中画一个类似 CLOCK 的图象， x 和 y 轴的范围都是从-1000 到+1000 。对于 MM_ANISOTROPIC ，也可以编类似的程序：

```
SetMapMode (hdc , MM_ANISOTROPIC) ;
SetWindowExtEx (hdc , 1000 , 1000 , NULL) ;
SetViewportExtEx (hdc , cxClient/2 , -cyClient/2 , NULL) ;
SetViewportOrgEx (hdc , cxClient/2 , cyClient/2 , NULL) ;
```

与 MM_ISOTROPIC 方式不同的是，这个钟一般是椭圆形的，而不是圆形的。

另一种使用 MM_ANISOTROPIC 的方法是将 x 和 y 轴的单位固定，但其值不相等。例如，如果有一个只显示文本的程序，你可能想根据单个字符的高度和宽度设置一种粗粒度的坐标：

```
SetMapMode (hdc , MM_ANISOTROPIC) ;
SetWindowExtEx (hdc , 1 , 1 , NULL) ;
SetViewportExtEx (hdc , cxChar , cyChar , NULL) ;
```

（这里假设 cxChar 和 cyChar 分别是一种字符间距固定的字体的字符宽度和高度，以象素为单位）。现在，你可以在 TextOut 调用中指定行和列坐标，而不是象素坐标。例如，下面的语句在距离左边 3 字符，上边 2 字符处显示文本 Hello:

```
TextOut (hdc , 3 , 2 , "Hello", 5) ;
```

这几乎是在非 Windows 的 MS_DOS 环境的文本方式下工作一样！

当你第一次设置 MM_ANISOTROPIC 映射方式时，它总是继承前面所设置的映射方式的范围，这会很方便。可以认为 MM_ANISOTROPIC 不“锁定”范围；也就是说，它允许你任意改变窗口范围。例如，假设你想用 MM_LOENGLISH 映射方式，因为希望逻辑单位为 0.01 英寸，但你不希望 y 轴的值向上增加，喜欢如 MM_TEXT 那样的方向，即 y 轴的值向下增加。可以使用如下的代码：

```
SIZE size ;
[其它程序行]
SetMapMode (hdc , MM_LOENGLISH) ;
SetMapMode (hdc , MM_ANISOTROPIC) ;

GetViewportExtEx (hdc , &size) ;

SetViewportExtEx (hdc , size.cx , -size.cy , NULL) ;
```

我们首先将映射方式设置为 MM_LOENGLISH ，然后，通过将映射方式设置为 MM_ANISOTROPIC 让范围可以自由改变。 GetViewportExtEx 获取视口范围并放到一个 SIZE 结构中，然后，我们使用范围来调用 SetViewportExtEx ，只是要将 y 范围取反。

4.6.7 WHATSIZE 程序

我们在下面四章中研究 GDI 函数时，将会使用这些映射方式。现在，我们来看一下以英寸和毫米为单位的客户区的大小。图 4-19 所示的 WHATSIZE 程序显示了客户区的大小，其单位分别与六个全约束的映射方式相关： MM_TEXT 、 MM_LOMETRIC 、 MM_HIMETRIC 、 MM_LOENGLISH 、 MM_HIENGLISH 和 MM_TWIPS 。

WHATSIZE.MAK

```
#------------------------
# WHATSIZE.MAK make file
#------------------------

whatsize.exe : whatsize.obj
     $(LINKER) $(GUIFLAGS) -OUT:whatsize.exe whatsize.obj $(GUILIBS)

whatsize.obj : whatsize.c
     $(CC) $(CFLAGS) whatsize.c
```

WHATSIZE.C

```
/*-----------------------------------------
   WHATSIZE.C -- What Size is the Window?
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "WhatSize" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

```
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "What Size is the Window?",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void Show (HWND hwnd, HDC hdc, int xText, int yText, int iMapMode,
           char *szMapMode)
     {
     char szBuffer [60] ;
     RECT rect ;

     SaveDC (hdc) ;

     SetMapMode (hdc, iMapMode) ;
     GetClientRect (hwnd, &rect) ;
     DPtoLP (hdc, (PPOINT) &rect, 2) ;

     RestoreDC (hdc, -1) ;

     TextOut (hdc, xText, yText, szBuffer,
              sprintf (szBuffer, "%-20s %7d %7d %7d %7d", szMapMode,
                   rect.left, rect.right, rect.top, rect.bottom)) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static char  szHeading [] =
                   "Mapping Mode            Left   Right     Top  Bottom" ;
     static char  szUndLine [] =
                   "------------            ----   -----     ---  ------" ;
     static int   cxChar, cyChar ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     TEXTMETRIC   tm ;

     switch (iMsg)
          {
```

```
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;

               ReleaseDC (hwnd, hdc) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               SelectObject (hdc, GetStpckObject (SYSTEM_FIXED_FONT)) ;

               SetMapMode (hdc, MM_ANISOTROPIC) ;
               SetWindowExtEx (hdc, 1, 1, NULL) ;
               SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;

               TextOut (hdc, 1, 1, szHeading, sizeof szHeading - 1) ;
               TextOut (hdc, 1, 2, szUndLine, sizeof szUndLine - 1) ;

               Show (hwnd, hdc, 1, 3, MM_TEXT,      "TEXT (pixels)") ;
               Show (hwnd, hdc, 1, 4, MM_LOMETRIC,  "LOMETRIC (.1 mm)") ;
               Show (hwnd, hdc, 1, 5, MM_HIMETRIC,  "HIMETRIC (.01 mm)") ;
               Show (hwnd, hdc, 1, 6, MM_LOENGLISH, "LOENGLISH (.01 in)") ;
               Show (hwnd, hdc, 1, 7, MM_HIENGLISH, "HIENGLISH (.001 in)") ;
               Show (hwnd, hdc, 1, 8, MM_TWIPS,     "TWIPS (1/1440 in)") ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 4-19 WHATSIZE程序

为了便于用 TextOut 函数显示信息， WHATSIZE 使用了 MM_ANISOTROPIC 映射方式，逻辑单位设置为字符的尺寸：

```
SetMapMode (hdc, MM_ANISOTROPIC) ;
SetWindowExtEx (hdc, 1, 1, NULL) ;
SetViewportExtEx (hdc, cxChar, cyChar, NULL) ;
```

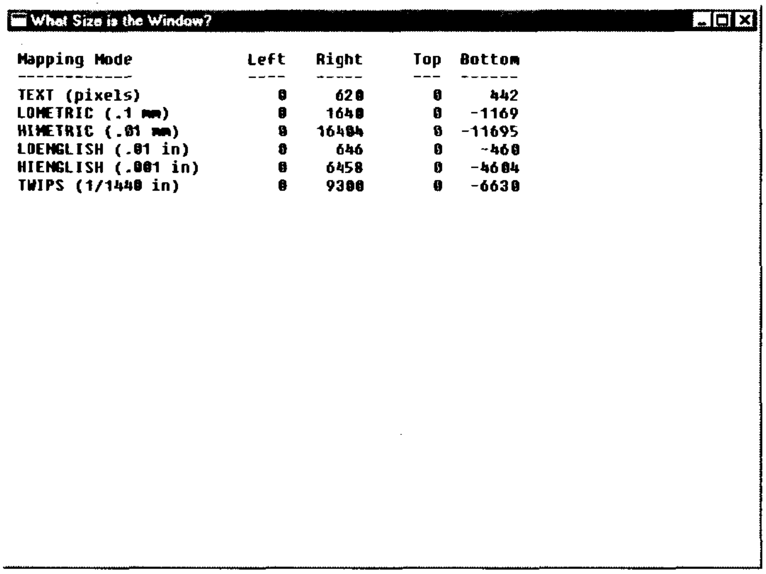
然后，程序可以使用固定间距字体的字符行和字符列坐标来指定 TextOut 调用的逻辑坐标。

当 WHATSIZE 需要获取某种映射方式下客户区的大小时，它保存当前的设备环境，设置一种新的映射方式，获取客户区坐标，将它们转换为逻辑坐标，然后在显示信息前，恢复原映射方式。这些代码在 WHATSIZE 的 Show 函数里：

```
SaveDC (hdc) ;
```

```
SetMapMode (hdc , iMapMode) ;
GetClientRect (hwnd , &rect) ;
DPtoLP (hdc , (PPOINT)&rect , 2) ;

RestoreDC (hdc , -1) ;
```

图 4-20 显示了 WHATSIZE 的一个典型输出。



Mapping Mode	Left	Right	Top	Bottom
TEXT (pixels)	0	620	0	442
LOMETRIC (.1 mm)	0	1640	0	-1169
HIMETRIC (.01 mm)	0	16404	0	-11695
LOENGLISH (.01 in)	0	646	0	-460
HIENGLISH (.001 in)	0	6458	0	-4604
TWIPS (1/1440 in)	0	9300	0	-6630

图 4-20 典型的WHATSIZE显示

你将注意到我使用了一种叫做 SYSTEM_FIXED_FONT 的东西来选择一种固定间距的字体。我将很快讨论它。

4.7 矩形、区域和剪裁

Microsoft Windows 95 包含了几种使用 RECT（矩形）结构和“区域”的绘图函数。一个区域是屏幕上的一块地方，它是矩形、多边形和椭圆的组合。

4.7.1 矩形函数

下面三个绘图函数需要一个指向矩形结构的指针：

```
FillRect (hdc, &rect, hBrush) ;
FrameRect (hdc, &rect, hBrush) ;
InvertRect (hdc, &rect) ;
```

在这些函数中， rect 参数是一个 RECT 类型的结构，它包含有 4 个域： left 、 top 、 right 和 bottom 。这个结构中的坐标被当作逻辑坐标。

FillRect 用指定画刷来填充矩形（直到但不包含 left 坐标和 bottom 坐标），该函数不需要先将画刷选进设备环境中。

FrameRect 使用画刷画矩形框，但是不填充矩形。使用画刷画矩形看起来有点奇怪，因为对于我们所介绍过的函数(如 Rectangle)，其边线都是用当前画笔绘制的。FrameRect 允许用户画一个不一定为纯色的矩形框。如果逻辑单位大于设备单位，则边界框将会为 2 个象素宽或者更宽。

InvertRect 将矩形中所有象素翻转，1 转换成 0，0 转换为 1，该函数将白色区域转变成黑色，黑色区域转变为白色，绿色区域转变成洋红色。

Windows 还提供了 9 个函数，使你可以更容易、更清楚地操作 RECT 结构。例如，要将 RECT 结构的四个域设置为特定值，通常使用如下的程序段：

```
rect.left = xLeft ;
rect.top = yTop ;
rect.right = xRight ;
rect.bottom = yBottom ;
```

但是，通过调用 SetRect 函数，只需要一条语句就可以得到同样的结果：

```
SetRect (&rect, xLeft, yTop, xRight, yBottom) ;
```

在你想要做以下事情之一时，可以很方便地选用其它八种函数：

- 将矩形沿 x 轴和 y 轴移动几个单元：

```
OffsetRect (&rect, x, y) ;
```

- 增减矩形的尺寸：

```
InflateRect (&rect, x, y) ;
```

- 矩形各域设置为 0

```
SetRectEmpty (&rect) ;
```

- 矩形拷贝给另一个矩形

```
CopypRect (&DestRect, &SrcRect) ;
```

- 获取两个矩形的交

```
IntersectRect (&DestRect, &SrcRect1, &SrcRect2) ;
```

- 获取两个矩形的并

```
UnionRect (&DestRect, &SrcRect1, &SrcRect2) ;
```

- 确定矩形是否为空

```
bEmpty = IsRectEmpty (&rect) ;
```

- 确定点是否在矩形内

```
bInRect = PtInRect (&rect, point) ;
```

在大多数情况下，这些函数的等价代码很简单。例如，你可以用下列语句来替代 CopyRect 函数调用：

```
DestRect = SrcRect ;
```

4.7.2 随机矩形

在图形系统中，有这么一个“永久”运行的有趣程序，它简单地使用随机的大小和色彩绘制一系列矩形。你可以在 Windows 中创建一个这样的程序，但是它并不象乍看起来那样容易编写。我希望你能认识到，你不能简单地在 WM_PAINT 消息中使用一个 while(TRUE) 循环。当然，它能够运行，但是程序将停止对其它消息的处理，同时，这个程序不能被中止或者被最小化。

一种可以接受的方法是设置一个 Windows 定时器，给窗口函数发送 WM_PAINT 消息（我将在第 7 章中讨论定时器）。对于每条 WM_PAINT 消息，你使用 GetDC 获取一个设备环境，画一个随机的矩形，然后用 ReleaseDC 释放设备环境。但是这样又降低了程序的趣味性，因为程序不能尽可能快地画随机矩形，它必须等待 WM_TIMER 消息，而这又依赖于系统时钟的分辨率。

在 Windows 中一定有很多“死时间”，在这个时间内，所有消息队列为空， Windows 只停在一个小循环中等待键盘或者鼠标输入。我们能否在死时间内获得控制，绘制矩形，并且只在有消息加入程序的消息队列之后才释放控制呢？这就是 PeekMessage 函数的目的之一。下面是 PeekMessage 调用的一个例子：

```
PeekMessage  (&msg, NULL, 0, 0, PM_REMOVE) ;
```

前面的四个参数（一个指向 MSG 结构的指针、一个窗口句柄、两个值指示消息范围）与 GetMessage 的参数相同。将第二、三、四个参数设置为 NULL 或 0 时，表明我们想让 PeekMessage 返回程序中所有窗口的所有消息。如果要将消息从消息队列中删除，则将 PeekMessage 的最后一个参数设置为 PM_REMOVE 。如果你不希望删除消息，那么你可以将这个参数设置为 PM_NOREMOVE 。这就是为什么 PeekMessage 是“偷看”而不是“获取”的原因，它使得程序可以检查程序的队列中的下一条消息，而不实际删除它。

GetMessage 不将控制返回给程序，直到从程序的消息队列中获取消息，但是 PeekMessage 总是立刻返回，而不论一个消息是否出现。当消息队列中有一个消息时，PeekMessage 的返回值为 TRUE （非零），并且将按通常方式处理消息。当队列中没有消息时， PeekMessage 返回 FALSE （ 0 ）。

这使得我们可以改写普通的消息循环。我们可以将如下所示的循环：

```
While (GetMessage (&msg, NULL, 0, 0))
     {
     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
     }
return msg 。 wParam ;
```

替换为下面的循环：

```
While (TRUE)
     {
     if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (msg.message == WM_QUIT)
               break ;

          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     else
          {
          [完成某些工作的其它程序行]
          }
     }
return msg.wParam ;
```

注意， WM_QUIT 消息是显式检查的。在普通的消息循环中你不必这么作，因为如果GetMessage 接收到一条 WM_QUIT 消息，它将返回 0，但是 PeekMessage 用它的返回值来指示是否得到一条消息，所以需要对 WM_QUIT 进行检查。

如果 PeekMessage 的返回值为 TRUE，则消息按通常方式进行处理。如果返回值为 FALSE，则在将控制返回给 Windows 之前，还可以作一点工作（如显示另一个随机矩形）。

（尽管 Windows 文档上说，你不能用 PeekMessage 从消息队列中删除 WM_PAINT 消息，但是这并不是什么大不了的问题。毕竟， GetMessage 并不从消息队列中删除 WM_PAINT 消息。从队列中删除 WM_PAINT 消息的唯一方法是令窗口客户区的失效区域变得有效，这可以用 ValidateRect 和 ValidateRgn 或者 BeginPaint 和 EndPaint 对来完成。如果你在使用 PeekMessage 从队列中取出 WM_PAINT 消息后，同平常一样处理它，那么就不会有问题了。所不能作的是使用如下所示的代码来清除消息队列中的所有消息：

```
while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE)) ;
```

这条语句从消息队列中移出并丢弃除了 WM_PAINT 之外的所有消息。如果队列中有一条 WM_PAINT 消息，程序就会永远地陷在 while 循环中。）

PeekMessage 在 Windows 的早期版本中比在 Windows 95 中要重要得多。这是因为 Windows 的 16 位版本使用的是非抢占的多任务（我将在第 14 章中讨论这一点）。 Windows 的 Terminal 程序在从通信线接收输入后，使用一个 PeekMessage 循环。打印管理器程序使用这个技术来进行打印，其它的 Windows 打印应用程序通常都会使用一个 PeekMessage 循环。在 Windows 95 可抢占的多任务环境下，程序可以创建多个线程，我们将第十四章看到这一点。

不管怎样，有了 PeekMessage 函数，我们就可以编写一个不停地显示随机矩形的程序。这个 RANDRECT 程序如图 4-21 中所示。

RANDRECT.MAK

```
#-----------------------
# RANDRECT.MAK make file
#-----------------------

randrect.exe : randrect.obj
     $(LINKER) $(GUIFLAGS) -OUT:randrect.exe randrect.obj $(GUILIBS)

randrect.obj : randrect.c
     $(CC) $(CFLAGS) randrect.c
```

RANDRECT.C

```
/*------------------------------------------
   RANDRECT.C -- Displays Random Rectangles
                 (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include <stdlib.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
void DrawRectangle (HWND) ;

int cxClient, cyClient ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "RandRect" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Random Rectangles",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (TRUE)
          {
          if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
               {
               if (msg.message == WM_QUIT)
                    break ;

               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          else
               DrawRectangle (hwnd) ;
          }

     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                          LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

void DrawRectangle (HWND hwnd)
     {
     HBRUSH hBrush ;
     HDC    hdc ;
     RECT   rect ;

     if (cxClient == 0 || cyClient == 0)
          return ;

     SetRect (&rect, rand () % cxClient, rand () % cyClient,
                     rand () % cxClient, rand () % cyClient) ;

     hBrush = CreateSolidBrush (RGB (rand () % 256, rand () % 256,
                                                    rand () % 256)) ;
     hdc = GetDC (hwnd) ;

     FillRect (hdc, &rect, hBrush) ;
```

```
ReleaseDC (hwnd, hdc) ;

DeleteObject (hBrush) ;
}
```

图 4-21 RANDRECT程序

程序使用我在上面讨论过的 SetRect 和 FillRect 函数，根据由 C 的 rand 函数得到的随机数决定矩形坐标和实心画刷的色彩。我将在第 14 章中提供这个程序的多线程版本。

4.7.3 创建和绘制区域

区域是对显示器上一个范围的描述，这个范围是矩形、多边形和椭圆的组合。区域可以用于绘制和剪裁，通过将区域选入设备环境，就可以用区域来进行剪裁（就是说，将可以绘图的范围限制为客户区的一部分）。与画笔、画刷和位图一样，区域是 GDI 对象，你应该调用 DeleteObject 来删除你所创建的区域。

当你创建一个区域时， Windows 返回一个该区域的句柄，类型为 HRGN 。最简单的区域是矩形，有两种创建矩形的方法：

```
hRgn = CreateRectRgn (xLeft, yTop, xRigth, yBottom) ;
```

或者

```
hRgn = CreateRectRgnIndirect (&rect) ;
```

你也可以创建椭圆区域：

```
hRgn = CreateEllipticRgn (xLeft, yTop, xRight, yBottom) ;
```

或者

```
hRgn = CreateEllipticRgnIndirect (&rect) ;
```

CreateRoundRectRgn 创建圆角的矩形区域。

创建多边形区域的函数类似于 Polygon 函数：

```
hRgn = CreatePolygonRgn (&point, iCount, iPolyFillMode) ;
```

point 参数是一个 POINT 类型的结构数组， iCount 是点的数目， iPolyFillMode 是 ALTERNATE 或者 WINDING 。你还可以用 CreatePolyPolygonRgn 来创建多个多边形区域。

那么，你会问，区域这东西究竟有什么特别之处？下面这个函数才显示出了区域的真正作用：

```
iRgnType = CombineRgn (hDestRgn, hSrcRgn1, hSrcRgn2, iCombine) ;
```

这一函数将两个区域（ hSrcRgn1 和 hSrcRgn2 ）组合起来并用句柄 hDestRgn 指向组合成

的区域。这三个区域句柄都必须是有效的，但是 hDestRgn 所指向的原来的区域被破坏掉了（当你使用这个函数时，你可能要让 hDestRgn 在初始时指向一个小的矩形区域）。

iCombine 参数说明 hSrcRgn1 和 hSrcRgn2 如何组合：

iCombine 值	新区域
RGN_AND	原来两个区域的公共部分
RGN_OR	原来两个区域的全部
RGN_XOR	原来两个区域的全部除去公共部分
RGN_DIFF	hSrcRgn1 的不在 hSrcRgn2 中的部分
RGN_COPY	hSrcRgn1 的全部

从 CombineRgn 返回的 iRgnType 值是下列之一： NULLREGION，表示得到一个空区域；SIMPLEREGION，表示得到一个简单的矩形、椭圆或者多边形； COMPLEXREGION，表示多个矩形、椭圆或多边形的组合； ERROR，表示出错了。

区域的句柄可以用于四个绘图函数：

```
FillRgn (hdc, hRgn, hBrush) :
FrameRgn (hdc, hRgn, hBrush, xFrame, yFrame) ;
InvertRgn (hdc, hRgn) ;
PaintRgn (hdc, hRgn) ;
```

FillRgn、 FrameRgn 和 InvertRgn 类似于 FillRect、 FrameRect 和 InvertRect。 FrameRgn 的 xFrame 和 yFrame 参数是画在区域周围的边框的宽度和高度。 PaintRgn 函数用设备环境中的当前画刷填充所指定的区域。所有这些函数都假定区域是用逻辑坐标定义的。

在你用完一个区域后，可以像删除其它 GDI 对象那样删除它：

```
DeleteObject (hRgn) ;
```

4.7.4 矩形与区域的剪裁

区域也在剪裁中扮演了一个角色。 InvalidateRect 函数使显示的一个矩形区域失效，并产生一个 WM_PAINT 消息。例如，你可以使用 InvalidateRect 函数来清除客户区并产生一个 WM_PAINT 消息：

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

你可以通过调用 GetUpdateRect 来获取失效矩形的坐标，并且可以使用 ValidateRect 函数使客户区的矩形有效。当你接收到一个 WM_PAINT 消息时，无效矩形的坐标可以从 PAINTSTRUCT 结构中得到，该结构是用 BeginPaint 函数填充的。这个无效矩形还定义了一个“剪裁区域”，你不能在剪裁区域外绘图。

Windows 有两个作用于区域而不是矩形的函数，它们类似于 InvalidateRect 和 ValidateRect：

```
InvalidateRgn (hwnd, hRgn, bErase) ;
ValidateRgn (hwnd, hRgn) ;
```

当你接收到一个由无效区域引起的 WM_PAINT 消息时，剪裁区域不一定是矩形。

你可以使用以下两个函数之一：

```
SelectObject (hdc, hRgn) ;
```

或者

```
SelectClipRgn (hdc, hRgn) ;
```

通过将一个区域选进设备环境中来创建自己的剪裁区域，这个剪裁区域使用设备坐标。

GDI 为剪裁区域建立一份拷贝，所以在将它选择到设备环境之后，用户可以删除它。Windows 还提供了几个对剪裁区域进行操作的函数，如 ExcludeClipRect 用于将一个矩形从剪裁区域里排除掉，IntersectClipRect 用于创建一个新的剪裁区域，它是前一个剪裁区域与一个矩形的交，OffsetClipRgn 用于将剪裁区域移动到客户区的另一部分。

4.7.5 CLOVER 程序

CLOVER 程序用四个椭圆组成一个区域，将这个区域选进设备环境中，然后画出从窗口客户区的中心出发的一系列直线，这些直线只出现在区域所限定的范围，结果显示如图 4-22 所示。

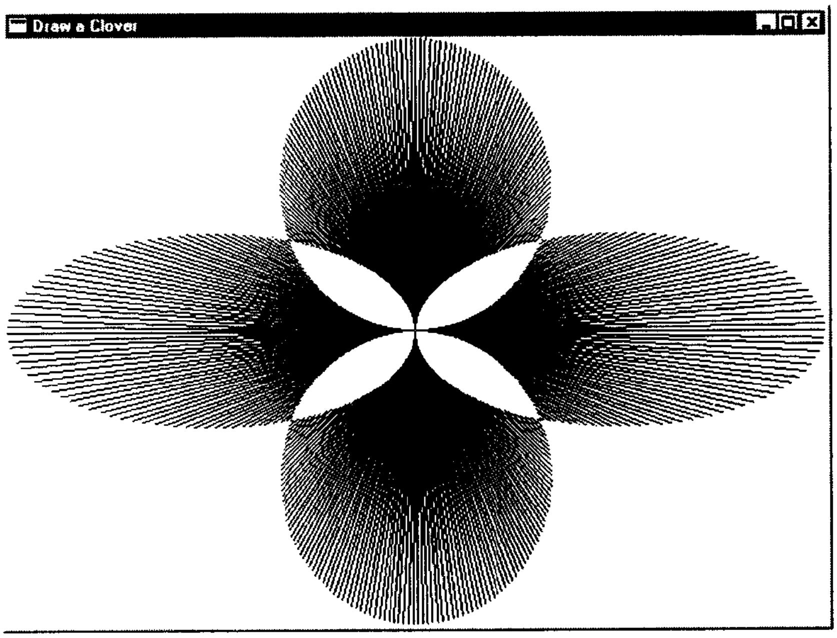


图 4-22 CLOVER利用复杂的剪裁区域画出的图形

要用常规的方法画出这个图形，就必须根据椭圆的边线公式计算出每条直线的端点。利用复杂的剪裁区域，可以直接画出这些线条，而让 Windows 确定其端点。CLOVER 程序如图 4-23 所示。

CLOVER.MAK

```
#----------------------
# CLOVER.MAK make file
#----------------------

clover.exe : clover.obj
     $(LINKER) $(GUIFLAGS) -OUT:clover.exe clover.obj $(GUILIBS)

clover.obj : clover.c
     $(CC) $(CFLAGS) clover.c
```

CLOVER.C

```
/*--------------------------------------------------
   CLOVER.C -- Clover Drawing Program using Regions
               (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>
#include <math.h>

#define TWO_PI (2.0 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Clover" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Draw a Clover",
                          WS_OVERLAPPEDWINDOW,
```

```
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static HRGN hRgnClip ;
     static int  cxClient, cyClient ;
     double      fAngle, fRadius ;
     HCURSOR     hCursor ;
     HDC         hdc ;
     HRGN        hRgnTemp [6] ;
     int         i ;
     PAINTSTRUCT ps ;

     switch (iMsg)
          {
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               hCursor = SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;

               if (hRgnClip)
                    DeleteObject (hRgnClip) ;
               hRgnTemp [0] = CreateEllipticRgn (0, cyClient / 3,
                                        cxClient / 2, 2 * cyClient / 3) ;
               hRgnTemp [1] = CreateEllipticRgn (cxClient / 2, cyClient / 3,
                                        cxClient, 2 * cyClient / 3) ;
               hRgnTemp [2] = CreateEllipticRgn (cxClient / 3, 0,
                                        2 * cxClient / 3, cyClient / 2) ;
               hRgnTemp [3] = CreateEllipticRgn (cxClient / 3, cyClient / 2,
                                        2 * cxClient / 3, cyClient) ;
               hRgnTemp [4] = CreateRectRgn (0, 0, 1, 1) ;
               hRgnTemp [5] = CreateRectRgn (0, 0, 1, 1) ;
               hRgnClip     = CreateRectRgn (0, 0, 1, 1) ;

               CombineRgn (hRgnTemp [4], hRgnTemp [0], hRgnTemp [1], RGN_OR) ;
               CombineRgn (hRgnTemp [5], hRgnTemp [2], hRgnTemp [3], RGN_OR) ;
               CombineRgn (hRgnClip,     hRgnTemp [4], hRgnTemp [5], RGN_XOR) ;

               for (i = 0 ; i < 6 ; i++)
                    DeleteObject (hRgnTemp [i]) ;
```

```
            SetCursor (hCursor) ;
            ShowCursor (FALSE) ;
            return 0 ;

       case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            SetViewportOrgEx (hdc, cxClient / 2, cyClient / 2, NULL) ;
            SelectClipRgn (hdc, hRgnClip) ;

            fRadius = _hypot (cxClient / 2.0, cyClient / 2.0) ;

            for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
                 {
                 MoveToEx (hdc, 0, 0, NULL) ;
                 LineTo (hdc, (int) ( fRadius * cos (fAngle) + 0.5),
                              (int) (-fRadius * sin (fAngle) + 0.5)) ;
                 }
            EndPaint (hwnd, &ps) ;
            return 0 ;

       case WM_DESTROY :
            DeleteObject (hRgnClip) ;
            PostQuitMessage (0) ;
            return 0 ;
       }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
  }
```

图 4-23 CLOVER程序

由于区域总是使用设备坐标，CLOVER程序必须在每次接收到WM_SIZE消息时重新创建区域，这可能需要几秒钟。CLOVER 从创建四个椭圆区域开始，这四个椭圆存放在hRgnTemp数组的头四个元素中，然后创建三个“哑”区域：

```
hRgnTemp[4] = CreateRectRgn (0, 0, 1, 1) ;
hRgnTemp[5] = CreateRectRgn (0, 0, 1, 1) ;
hRgnClip    = CreateRectRgn (0, 0, 1, 1) ;
```

客户区左右的两个区域被组合起来：

```
CombineRgn (hRgnTemp[4], hRgnTemp[0], hRgnTemp[1], RGN_OR) ;
```

同样，客户区上下两个椭圆区域被组合起来：

```
CombineRgn (hRgnTemp[5], hRgnTemp[2], hRgnTemp[3], RGN_OR) ;
```

最后，两个组合后的区域再被组合到hRgnClip中：

```
Combine (hRgnClip, hRgnTemp[4], hRgnTemp[5], RGN_OR) ;
```

RGN_OR 用于从结果区域中排除重叠部分。最后，删除6个临时区域：

```
for (i=0 ; i<6 ; i++)
    DeleteObject (hRgnTemp[i]) ;
```

与画出的图形比起来， WM_PAINT 的处理很简单。视口原点被设置为客户区的中心（使画直线更容易一些），在 WM_SIZE 消息期间创建的区域被选择为设备环境的剪裁区域：

```
SetViewportOrg (hdc, xClient/2, yClient/2) ;
SelectClipRgn (hdc, hRgnClip) ;
```

现在，剩下的就是画直线了，共360条，每隔一度画一条。每条线的长度为变量 fRadius，这是从中心到客户区的角落的距离：

```
fRadius = _hypot (xClient/2.0, yClient/2.0) ;

for (fAngle = 0.0 ; fAngle < TWO_PI ; fAngle += TWO_PI / 360)
    {
    MoveToEx (hdc, 0, 0, NULL) ;
    LineTo (hdc, (int) ( fRadius * cos (fAngle) + 0.5),
                 (int) (-fRadius * sin (fAngle) + 0.5)) ;
    }
```

在处理 WM_DESTROY 消息时，删除该区域：

```
DeleteObject (hRgnClip) ;
```

4.8 路　径

路径是存储在 GDI 内部的一些直线和曲线的集合，它首先被引入了 Windows NT，并且也为 Windows 95 所支持。粗看起来，路径和区域可能非常相似，事实上，你也可以将路径转换到区域，并且，可以用路径来剪裁。不过，我们很快就会看到，它们究竟有何不同。

4.8.1 创建和绘制路径

路径的定义从下面的调用开始：

```
BeginPath (hdc) ;
```

这一调用之后，你所画的任何线条都被作为路径存储到 GDI 内部，而不在设备环境中绘制出来。通常，路径由首尾相接的线条构成，要画出这样的线条，可以使用 LineTo 、PolylinTo 和 BezierTo 等函数，它们都从当前位置开始画线。如果你用 MoveToEx 改变了当

前位置，或者如果你调用了任何其它的画线函数，或者调用了某个改变当前位置的窗口/视口函数，那你就在整个路径中创建了一条新的子路径。由此可见，一条路径包含一条或多条子路径，而每一条子路径是一系列首尾相接的线条。

路径中的每一子路径可以是开放的或封闭的。封闭子路径的第一条线的第一个点与最后一条线的最后一个点相同，而且，该子路径由调用 CloseFigure 结束。如果必要的话，CloseFigure 用一条直线使子路径封闭，此后的画线函数将开始新的一条子路径。最后，通过如下调用来结束路径定义：

```
EndPath (hdc) ;
```

此后，你可以调用下面的五个函数之一：

```
StrokePath (hdc) ;
FillPath (hdc) ;
StrokeAndFillPath (hdc) ;
hRgn = PathToRegion (hdc) ;
SelectClipPath (hdc, iCombine) ;
```

在这些函数的处理结束后，路径定义都将被破坏。

StrokePath 使用当前画笔绘制路径。你可能会感到奇怪：这是干什么？我干嘛不把路径之类的东西抛在一边，直接像通常那样画线呢？我很快会告诉你为什么。

另外四个函数都会用直线将开放的路径封闭起来。FillPath 用当前画刷，根据当前多边形填充模式来填充路径。StrokeAndFillPath 同时完成前面两个函数的功能。你还可以将路径转换成区域或者将路径用作剪裁范围。iCombine 参数是 CombineRgn 函数所使用的 RGN_常量之一，它指明路径如何与当前剪裁区域组合。

由于区域只能由矩形、椭圆和多边形的组合来定义，所以对于填充和剪裁而言，路径具有更大的灵活性。路径可以由 Bezier 样条和（至少在 Windows NT 中）弧组成。在 GDI 中，路径与区域的存储形式是很不一样的，路径由直线和曲线的定义构成，而区域（一般来说）由扫描线构成。

4.8.2 扩展的画笔

在调用 StrokePath 时，路径是用当前画笔绘制的。本章的前面，我们曾经讨论过 CreatePen 函数，它可以用来创建画笔对象。有了路径之后，Windows NT 和 Windows 95 还支持一种扩展的画笔函数调用 ExtCreatePen，这个函数说明了为何有时先创建路径再把它画出来比不用路径而直接画出那些线条更有用。ExtCreatePen 函数如下：

```
hPen = ExtCreatePen (iStyle, iWidth, &lBrush, 0, NULL) ;
```

可以将此函数用于普通的线条绘制，不过，那样的话，Windows 95 不支持它的部分特性。即使用于绘制路径，它的某些特性仍然不为 Windows 95 所支持。我将上面的函数调用中后两个参数设为 0 和 NULL 来表明了这一点。

ExCreatePen 的第一个参数可以是 CreatePen 所用的任何一种风格。你还可以将这些风格与 PS_GEOMETRIC 或 PS_COSMETIC 组合起来使用。在前一种组合情况下， iWidth 参数用逻辑单位来定义线条的宽度，并且可能受变换的影响；在后一种组合情况下， iWidth 参数必须是 1 。在 Windows 95 中，短线(dashed)和点线(dotted)风格必须是 PS_COSMETIC （ Windows NT 中没有这一限制）。

CreatePen 的参数之一是色彩；与此不同， ExtCreatePen 使用画刷来着色 PS_GEOMETRIC 画笔的内部，画刷甚至可以用位图来定义。

当你画较宽的线条时，你可能还会关心线条端点的样子。如果线条或曲线首尾相接，你可能还会关心这些线条结合处的样子。使用 CreatePen 所创建的画笔，端点和结合点总是圆的，而 ExtCreatePen 为你提供了选择的可能（实际上，在 Windows 95 中，只有当你用这样的画笔来画路径时，才能做此选择； Windows NT 要灵活得多）。可以用下列画笔风格之一来调用 ExtCreatePen ，以定义较宽线条的端点形状：

```
PS_ENDCAP_ROUND
PS_ENDCAP_SQUARE
PS_ENDCAP_FLAT
```

“方块(square)”风格与“平坦(flat)”风格不同，它多画出线条的一半宽度。类似地，路径中线条的连接点形状可如下定义：

```
PS_JOIN_ROUND
PS_JOIN_BEVEL
PS_JOIN_MITER
```

“斜切(bevel)”风格截去连接点，“斜接(miter)”风格把连接点变成楔形。下面的 ENDJOIN 程序更清楚地表明了这些值的含义。

ENDJOIN.MAK

```
#------------------------
# ENDJOIN.MAK make file
#------------------------

endjoin.exe : endjoin.obj
     $(LINKER) $(GUIFLAGS) -OUT:endjoin.exe endjoin.obj $(GUILIBS)

endjoin.obj : endjoin.c
     $(CC) $(CFLAGS) endjoin.c
```

ENDJOIN.C

```
/*----------------------------------------
   ENDJOIN.C -- Ends and Joins Demo
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "EndJoin" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (WNDCLASSEX) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Ends and Joins Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static int  iEnd  [] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE,
                              PS_ENDCAP_FLAT } ;
     static int  iJoin [] = { PS_JOIN_ROUND,   PS_JOIN_BEVEL,
                              PS_JOIN_MITER } ;
     static int  cxClient, cyClient ;
     HDC         hdc ;
     int         i ;
     LOGBRUSH    lb ;
     PAINTSTRUCT ps ;

     switch (iMsg)
          {
```

```
          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               SetMapMode (hdc, MM_ANISOTROPIC) ;
               SetWindowExtEx (hdc, 100, 100, NULL) ;
               SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;

               lb.lbStyle = BS_SOLID ;
               lb.lbColor = RGB (128, 128, 128) ;
               lb.lbHatch = 0 ;


               for (i = 0 ; i < 3 ; i++)
                    {
                    SelectObject (hdc,
                         ExtCreatePen (PS_SOLID | PS_GEOMETRIC |
                                       iEnd [i] | iJoin [i], 10,
                                       &lb, 0, NULL)) ;

                    BeginPath (hdc) ;

                    MoveToEx (hdc, 10 + 30 * i, 25, NULL) ;
                    LineTo   (hdc, 20 + 30 * i, 75) ;
                    LineTo   (hdc, 30 + 30 * i, 25) ;


                    EndPath (hdc) ;

                    StrokePath (hdc) ;

                    DeleteObject (
                         SelectObject (hdc,
                              GetStockObject (BLACK_PEN))) ;

                    MoveToEx (hdc, 10 + 30 * i, 25, NULL) ;
                    LineTo   (hdc, 20 + 30 * i, 75) ;
                    LineTo   (hdc, 30 + 30 * i, 25) ;
                    }
               EndPaint (hwnd, &ps) ;
               return 0 ;

               case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }

     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 4-24 ENDJOIN程序

ENDJOIN 程序画三个 V 形的宽线条，按前面列出的顺序分别使用各种端点和连接点风格。它还使用现有画笔画了三条完全相同的线条，以便在宽线条和正常线条之间做个对比。结果如图 4-25 所示。

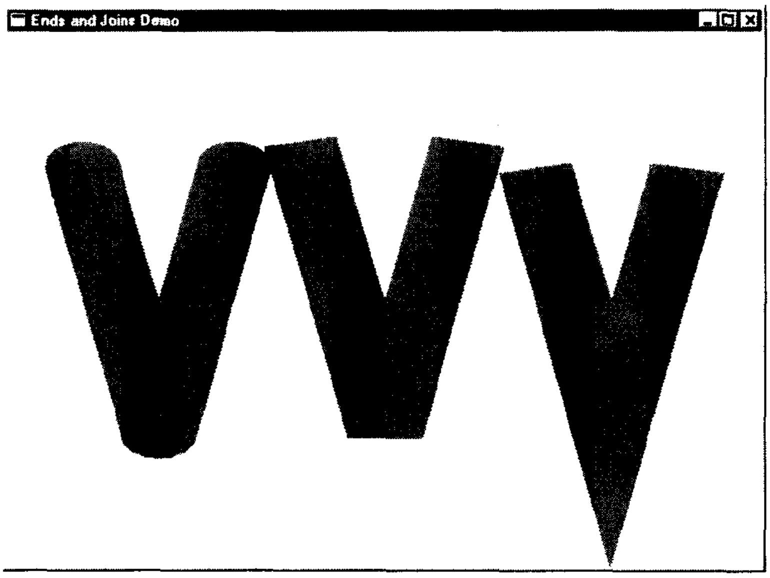


图 4-25 ENDJOIN的显示

我想，Windows 95 支持 StrokePath 函数的理由现在应该清楚了：如果你分别绘制这两条线，GDI 只能将连接点视为端点；而只要这两条线在一条路径中，GDI 就会知道它们是连接在一起的，从而正确地决定连接点。

4.9 BITS 与 BLTS

位图代表了 Windows 95 程序中存储图象信息的两种方法之一，它是图象的数字化表示，图象中的每个象素对应于位图中的一个或多个位。单色位图中的每个象素只需要一位；彩色位图则需要更多位，以指示每个象素的色彩。存储图象信息的另一种形式是元文件，我将在本章后面加以介绍。元文件是对图象的描述而不是数字化表示。

在计算机图形学中，位图和元文件各有其作用。位图经常用来表示来自真实世界的复杂图象，如数字化的照片或者捕获的视频图象。元文件更适合于描述由人或者计算机生成的图象，如建筑图纸等。位图和元文件都能存在于存储器中，或者作为文件存放在磁盘上，两者都可以通过剪贴板在 Windows 应用程序之间传送。

可以利用 Windows 95 提供的 Paint 程序“手工”构造位图，然后，将位图作为资源包含到资源描述文件中，并用 LoadBitmap 函数将它载入程序，这些是我们将在第 9 章中看到的。在第 10 章中，我们将看到位图是如何代替菜单中的文本。位图还可以用来构造定制的画刷。

位图有两个主要的不足。首先，它们极容易受到设备相关性之类问题的影响。最明显的与设备相关的特性是色彩。在单色设备上显示彩色位图通常是不会令人满意的。另一个

问题是，位图隐含着特定的分辨率和图象的纵横比。尽管位图可以延展或者压缩，但是这一过程通常涉及到重复或者删除象素的行或列，从而引起缩放后的图象失真。元文件几乎可以缩放到任意尺寸，而不会导致失真。

位图的第二个主要缺陷是，它需要的存储空间太大。例如，表示一个完整的 640 × 480 象素， 16 色 VGA 屏幕上的图形需要超过 150KB 的空间。元文件需要的存储空间通常比位图少得多。位图的存储空间是由图象的大小和它所包含的色彩数目所决定的，而元文件的存储空间是由图象的复杂程度和它所包含的单个 GDI 指令的数目决定的。

位图较元文件的优越之处在于速度。将位图拷贝到视频显示器上比起重新执行元文件总是要快得多。

4.9.1 色彩和位图

图象中的每个象素对应于位图中的一个或者多个位。单色图象中，每个象素只需要一位；彩色图象中，每个象素需要多位。一个位图可以表示的不同色彩数等于以 2 为底数，以每象素位数为指数计算出来的幂。例如， 16 色位图中的每个象素需要 4 位， 256 色位图中的每个象素需要 8 位，“全彩色”位图中，每个象素需要 24 位，其中每 8 位分别代表红、绿和蓝。

在 Windows 3.0 之前， Windows 所支持的唯一位图是 GDI 对象，它可以用位图句柄来引用。这些位图是单色的，或者与真实图形输出设备（如视频显示器）有同样的色彩组织。例如，一个与 16 位 VGA 兼容的位图有四个色彩平面。问题在于，这些彩色位图不能被保存下来以在具有不同色彩组织的图形输出设备上使用，比如每个象素使用 8 位，能显示 256 种色彩的设备。

从 Windows 3.0 开始，定义了一种新的位图格式，叫做设备无关位图，简写为 DIB 。DIB 包含它自己的色彩表，它说明象素位如何与 RGB 色彩相对应。 DIB 可以在任意光栅输出设备上显示。唯一的问题是， DIB 的色彩必须被转换成设备所能表示的最相近的色彩。

4.9.2 设备无关位图（ DIB)

DIB 格式被叫做“设备无关的”，因为它包含有一个色彩表，色彩表描述了象素如何与 RGB 色彩值相对应，这个色彩表不必与特定图形输出设备兼容。 DIB 格式是 OS/2 1.1 Presentation Manager（显示管理器）所支持的位图格式的扩展， Windows 头文件中包含一些用于 OS/2 位图的结构。

由于 DIB 的引入，有时称 GDI 位图对象为“设备有关位图”（ DDB ）。之所以说它们与设备有关，是因为它们必须与某种特定的图形输出设备兼容。 DIB 不是 GDI 对象，GDI 不能存储 DIB ， DIB 必须存储在 GDI 之外的文件中或者全局存储块中，一旦 GDI 获取一个 DIB 之后它就会将该 DIB 转化成与设备相关的位图，以便与某种真实设备兼容。DIB 主要用于程序之间的交换，通过将 DIB 存储到文件中，或者将它们拷贝到剪贴板上，均可以实现程序之间的传送。

DIB 可以转换成与设备相关的 GDI 位图对象，在这种情况下， DIB 与设备无关的色彩

信息就丢失了。 GDI 位图对象也能用于构造 DIB ，在这种情况下， DIB 将包含一个与图形输出兼容的色彩表，而 GDI 位图对象也与该输出设备兼容。

如果你需要以一种与设备无关的格式将位图信息存放到文件中，读取位图文件或者将位图信息传送进/出剪贴板，就必需使用 DIB 。但是，如果你只需要创建或者使用单色位图，或者只需要在自己的程序中使用与视频显示器兼容的位图，则使用与设备有关的 GDI 位图对象通常要容易得多。

4.9.3 DIB 文件

你可以用零售的 Windows 95 产品中提供的 Microsoft Developer Studio 或者 Paint 程序来创建设备无关位图，并将它保存到磁盘文件中。文件通常具有扩展名.BMP ，但是也有一些 DIB 使用扩展名.DIB 。

DIB 文件以一个文件头部分开始，它由 BITMAPFILEHEADER 结构定义，这个结构有五个字段：

字段	长度	描述
bfType	WORD	两字节的 “ BM ” （代表位图）
bfSize	DWORD	文件的总长度
bfReserved1	WORD	设置为 0
bfReserved2	WORD	设置为 0
bfOffBits	DWORD	位图的二进制位离文件头的偏移地址

后跟另一个文件头部分，它由 BITMAPINFOHEADER 结构定义，这个结构有 11 个字段：

字段	长度	描述
biSize	DWORD	这个结构的长度，以字节为单位
biWidth	LONG	位图的宽度，单位为象素
biHeight	LONG	位图的高度，单位为象素
biPlanes	WORD	设置为 1
biBitCount	WORD	每象素所需要的色彩位（ 1 、 4 、 8 或者 24)
biCompressioin	DWORD	压缩方法（ 0 代表不压缩）
biSizeImage	DWORD	位图的大小，以字节为单位（只有在使用压缩时才需要）
biXPelsPerMeter	LONG	水平分辨率，单位为象素/米
biYPelsPerMeter	LONG	垂直分辨率，单位为象素/米
biClrUsed	DWORD	图象中使用的色彩数
biClrImportant	DWORD	图象中重要色彩的数目

在 biBitCount 字段之后的所有字段都可能以 0 为缺省值（或者甚至不在文件中出现）。因此，结构的长度可以小至只有 16 字节。这个结构也可以在表中所列字段之外增加其

它字段。

如果 biClrUsed 被设置为 0，并且每象素的色彩位数为 1 、 4 或者 8，则 BITMAPINFOHEADER 结构后跟一色彩表，色彩表由两个或者更多的 RGBQUAD 结构组成。RGBQUAD 结构定义了一个 RGB 色彩值：

字段	长度	描述
rgbBlue	BYTE	蓝色的强度
rgbGreen	BYTE	绿色的强度
rgbRed	BYTE	红色的强度
rgbReserved	BYTE	设置为 0

RGBQUAD 结构的数目通常由 biBitCount 字段确定：1 个色彩位需要两个 RGBQUAD 结构，4 个色彩位需要 16 个 RGBQUAD 结构，8 个色彩位需要 256 个 RGBQUAD 结构。但是，如果 biClrUsed 字段不为 0，则 biClrUsed 字段包含了色彩表中 RGBQUAD 结构的数目。

色彩表后面是定义位图图象的二进制位数组，这个数组从底行象素开始，每行从最左边的象素开始，每个象素相应于 1 、 4 、 8 或者 24 位。对于每个象素一个色彩位的单色位图，每行的每一个象素由每行第一个字节的最高位表示。如果这一位为 0，则象素的色彩可以从色彩表中的第一个 RGBQUAD 结构中获取。如果该位为 1，则色彩由色彩表中的第二个 RGBQUAD 结构给出。

对于一个 16 色位图（每个象素 4 个色彩位），每行的第一个象素由每行第一个字节中最高的 4 位来表示。每个象素是通过用 4 位值作为色彩表中 16 个表项的索引来获取的。

对于 256 色的位图，每个字节对应于一个象素，象素的色彩用这个 8 位值作为色彩表的 256 个表项的索引来获取。

如果位图图象中每象素为 24 个色彩位，每三个字节是一个象素的 RGB 值，在这种情况下将没有色彩表，除非 BITMAPINFOHEADER 结构的 bitClrUsed 字段不为 0。

每种情况下，位图数据的每行包含 4 的倍数个字节，该行被填充至右边以确保这一点。

OS/2 1.1 及其以后的版本所支持的位图格式是很相似的，它以一个 BITMAPFILEHEADER 结构开始，不过，它后面是一个 12 字节的 BITMAPINFOHEADER 结构（你可以通过查看该结构的第一个字段来确定位图文件是使用这种格式还是使用 Windows 格式）。色彩表由 RGBTRIPLE 结构而不是 RGBQUAD 结构组成。

从 Windows 95 开始，定义了另外一种信息头，叫做 BITMAPV4HEADER（Windows 95 有时也被称作为 Windows 4.0，所以有 V4 或者“版本 4”这样的叫法）。这个头中包含一些用于在不同设备上正确地显示色彩所需要的信息。

4.9.4 紧缩的 DIB 存储格式

如果你遇到一个 DIB 文件，想要将它读入 Windows 程序中，那么你可以直接将它读进一个全局存储块中，这被称为“紧缩 DIB 存储格式”，它包含 DIB 文件中除了

BITMAPFILEHEADER 结构之外的所有数据。因此，一个包含 DIB 的全局存储块以一个 BITMAPINFOHEADER 结构开始，后跟一个色彩表（如果有的话），再跟位图位。紧缩的 DIB 存储格式用于将 DIB 拷贝到剪贴板或者将 DIB 从剪贴板拷贝下来。

你也可以使用 SetDIBBitsToDevice 或者 StretchDIBBits 函数来将缩减的 DIB 存储格式在屏幕上显示出来、创建一个基于 DIB 的画刷（用 CreateDIBPatternBrush)或者根据 DIB 创建一个与设备相关的 GDI 位图对象（用 CreateDIBitmap ）。

每种情况下，你都必须计算位图位于全局存储块中的开始地址。你可以通过检查信息头结构，确定色彩表的大小来做到这一点。

4.9.5 显示 DIB

Windows 有两个函数，它们能用缩减的 DIB 存储块显示 DIB 。最常用的是 StretchDIBites ，它将位图放大或缩小，并且可以使用本章后面讨论的各种光栅操作。SetDIBitsToDevice 函数要简单一些，它显示位图，不放缩，也不能使用任何光栅操作。

两个函数都要求一个指向 DIB 存储块起点的 BITMAPINFOHEADER 结构的指针和一个指向位图二进制位的指针。 StretchDIBits 函数很灵活，它允许你以指定的逻辑宽度和高度在显示器上显示位图内的任何矩形部分。

4.9.6 将 DIB 转换为位图对象

如果你使用 16 色或者 256 色的视频显示器显示一个全彩色位图，其中每象素 24 位，你将注意到位图的显示需要一些时间，这是因为设备驱动程序必须对位图中每个象素执行最相近色彩搜索。

通过 CreateDIBitmap 函数将 DIB 转换成与设备相关的 GDI 位图对象,可以提高显示处理的速度。最相近色彩的搜索只需执行一次，然后位图对象将与视频显示具有相同的格式。

与这个函数的名字相反， CreateDIBitmap 并不创建 DIB ,它用一个 DIB 说明创建与设备相关的 GDI 位图对象，并返回该对象的句柄。这个 GDI 位图对象与图形输出设备兼容，设备的句柄作为第一个参数传送给函数。与显示 DIB 时相似， GDI 必须将设备无关的色彩转换为设备的色彩。

在这个调用之后，通过将位图句柄选进内存设备环境，并使用 BitBlt （如本章后面所示）就能显示位图。

你也可以使用 CreateDIBitmap 创建一个未初始化的 GDI 位图对象：

```
hBitmap = CreateDIBitmap (hdc, &bmih, 0, NULL, NULL, 0) ;
```

有两个函数可用于设置和获取位图的位。第一个函数设置位：

```
SetDIBits (hdc, hBitmap, iStart, iNum, pBits, &bmi, iUsage) ;
```

后面三个参数与 CreateDIBitmap 函数的相应参数相同。 nStart 参数指定 pBits 所指向的位图

的起始扫描行，它可以从 0（代表最下面的扫描行）至位图的高度（单位为象素）减 1（代表最上面的扫描行）。iNum 参数指示设置到位图中的扫描行数。

GetDIBits 函数具有同样的参数：

```
GetDIBits (hdc, hBitmap, iStart, iNum, pBits, &bmi, iUsage) ;
```

但是在这种情况下，pBits 指向接收位图二进制位的缓冲区，函数设置 BITMAPINFO 结构的字段，以指示位图和色彩表的大小。

4.10 GDI 位图对象

源于 Windows 1 的早期位图格式限制很多，它高度依赖于位图为之创建的输出设备。在磁盘上存储位图文件时，你应该使用 DIB 格式，而不应该使用早期位图格式。尽管如此，如果所需位图只在程序内使用，那么采用与设备有关的位图要容易得多，性能也会更好。

4.10.1 在程序中创建位图

Windows 提供了五个函数，使你可以在程序中创建与设备有关的 GDI 位图对象。第一种是上面讨论的 CreateDIBitmap 函数，其它函数是：

```
hBitmap = CreateBitmap (cxWidth, cyHeight, iPlanes, iBitsPixel, pBits) ;
hBitmap = CreateBitmapIndirect (&bitmap) ;
hBitmap = CreateCompatibleBitmap (hdc, cxWidth, cyHeight) ;
hBitmap = CreateDiscardableBitmap (hdc, cxWidth, cyHeight) ;
```

在这些函数中，cxWidth 和 cyHeight 参数都是位图的宽度和高度（单位为象素）。在 CreateBitmap 中，nPlanes 和 nBitsPixel 参数分别是色彩平面数和位图中每象素的色彩位数，两个参数中至少有一个为 1。如果两个参数都为 1，则函数创建的是单色位图（我马上将讨论色彩平面和色彩位是如何表示色彩的）。

如果你要创建的是未初始化的位图，可以将 CreateBitmap 函数中的 pBits 设置为 NULL，所得位图包含随机数据。在 CreateCompatibleBitmap 和 CreateDiscardableBitmap 函数中，Windows 使用 hdc 所指示的设备环境来获取色彩平面数和每象素的色彩位数，这些函数创建的位图都未初始化。

CreateBitmapIndirect 与 CreateBitmap 类似，只是它使用 BITMAP 类型的位图结构来定义位图。下表显示了这个结构的各个字段：

字段	类型	描述
bmType	LONG	设置为 0
bmWidth	LONG	位图宽度，以象素为单位
bmHeight	LONG	位图高度，以扫描行为单位

续表

字段	类型	描述
bmWidthBytes	LONG	位图宽度，以字节为单位（必须为偶数）
bmPlanes	WORD	色彩平面数
bmBitsPixel	WORD	每象素色彩位数
bmBits	LPVOID	指向位数组的指针

bmWidthBytes 字段必须是偶数，它表示存储一个扫描行所需的最小偶字节数。bmBites 所指示的位数组必须根据 bmWidthBytes 字段来组织。如果 bm 是一个类型为 BITMAP 的结构变量，那么可以用下面的语句来计算 bmWidthBytes 字段：

```
bm.bmWidthBytes = (bm.bmWidth * bm.bmBitsPixel + 15) / 16 * 2 ;
```

如果 Windows 不能创建位图（一般是因为没有足够的存储），那么它将返回 NULL。你必须检查从位图创建函数返回的值，特别是在创建大位图时。

位图创建之后，就不能改变其大小、色彩平面数或者象素的色彩位数。为此，你必须另外创建一个位图，并将原位图的位传送给新的位图。如果你有一个位图句柄，就可以使用如下的语句来获取尺寸和色彩结构：

```
GetObject (hBitmap, sizeof (BITMAP), (LPVOID)&bitmap) ;
```

这样就将位图信息拷贝到了一个类型为 BITMAP 的结构（在这里为 bitmap）中。这个函数不填充 bmBits 字段，为了访问位图的实际位，必须调用：

```
GetBitmapBits (hBitmap, dwCount, pBits) ;
```

这样将 dwCount 位拷贝到指针 pBits 指向的一个字节数组中。为了确保位图的所有位都拷贝进这个数组，可以根据位图结构的字段计算 dwCount 参数：

```
dwCount = (DWORD)bitmap.bmWidthBytes * bitap.bmHeight * bitmap.bmPlanes ;
```

你也可以使用以下的函数指示 Windows 将一个包含有位图位的字符数组拷贝到已经存在的位图中：

```
SetBitmapBits (hBitmap, dwCount, pBits) ;
```

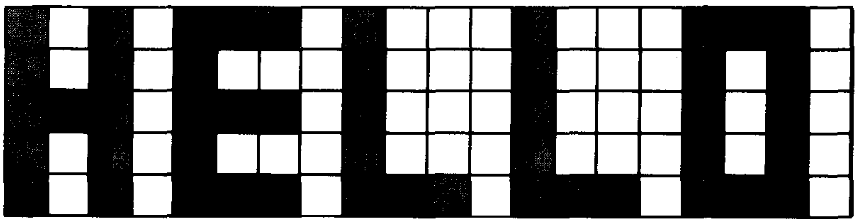
由于位图是 GDI 对象，所以必须删除所创建的位图：

```
DeleteObject (hBitmap) ;
```

4.10.2 单色位图格式

单色位图的位格式要相对简单一些，几乎可以直接从你所要创建的图象中推出来。例

如，假设你想创建一个外形如下所示的位图：



你可以写下一系列的位（0代表黑，1代表白），直接对应于这个网格。从左到右读这些位，然后，以每8位一组指定一个十六进制字节。如果位图的宽度不是16的整数倍，则在字节的右部填充0，以得到偶数个字节：

```
0 1 0 1 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0 1 = 51 77 10 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 0 0 1 0 0 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 13 77 50 00
0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 = 57 77 50 00
0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 = 51 11 10 00
```

这个位图的相应宽度（以象素为单位）为20，高度为5个扫描行，字节宽度为4。可以用下面的语句为位图建立一个BITMAP结构：

```
static BITMAP bitmap = {0, 20, 5, 4, 1, 1}
```

并且可以将这些二进制位存放到一个BYTE数组中：

```
static BYTE byBits[]= {0x51, 0x77, 0x10, 0x00,
                       0x57, 0x77, 0x50, 0x00,
                       0x13, 0x77, 0x50, 0x00,
                       0x57, 0x77, 0x50, 0x00,
                       0x51, 0x11, 0x10, 0x00 } ;
```

用CreateBitmapIndirect创建位图需要两条语句：

```
bitmap.bmBits = (LPVOID)byBits ;
hBitmap = CreateBitmapIndirect (&bitmap) ;
```

另一种方法是：

```
hBitmap = CreateBitmapIndirect (&bitmap) ;
SetBitMapBits (hBitmap, sizeof (byBits), byBits) ;
```

你也可以使用一条语句来创建位图：

```
hBitmap = CreateBitmap (20, 5, 1, 1, byBits) ;
```

4.10.3 彩色位图格式

旧式风格的彩色位图更复杂一些并且非常依赖于设备，它的组织使得将每一位传输到特定的输出设备很容易。位图是组织为一系列色彩平面还是组织为每象素含多个色彩位取决于位图所相适合的设备。

我们先来看看 bmBitsPixel 值为 1（这意味着每象素具有 1 个色彩位）而 bmPlanes 值大于 1 的位图。16 色 VGA 是一个好例子，Windows 利用 VGA 的四个色彩平面来显示 16 种色彩，因此 bmPlanes 为 4 。位数组从最上面的扫描行开始，每个扫描行的色彩平面是顺序存储的——先存放红色平面，然后是绿色平面、蓝色平面，最后是亮度平面，接着是位图的第二个扫描行。

位图也可以每象素用多个位来表示色彩。假设一种视频设备可以使用 8 个色彩位（一个字节）来表示 256 种色彩，对于每个扫描行，第一个字节代表最左象素的色彩，第二字节表示下一个象素的色彩，依此类推。BITMAP 结构中的 bmWidthBytes 值反映了每个扫描行填充后的字节宽度，但是 bmWidth 值仍然是每个扫描行的象素的数目。

问题在于：位图中没有任何信息指定多个色彩平面或者多个色彩位是如何对应于实际的色彩的。特定的彩色位图只适合于在那些存储组织与位图相同的输出设备上显示，因此，我们不推荐用 CreateBitmap 或者 CreateBitmapIndirect 来创建已初始化的彩色位图。你应该只将这些函数用于创建已初始化的或者未初始化的单色位图。

要创建彩色位图，应该使用 CreateCompatibleBitmap ，它能保证格式与真实图形输出设备兼容。通过将位图选进内存设备环境（马上要讲到）中并在该设备环境上画出位图，或者通过使用 BitBlt 函数，都可以将图象变成彩色位图。

要创建不必与真实图形输出设备兼容的彩色位图，可以使用 DIB 。

4.10.4 内存设备环境

SetDIBitsToDevice 和 StretchDIBits 这两个函数允许你在输出设备上显示二进制位数组，但是，即使你有一个位图句柄，却不会有函数将该位图输出到设备环境的显示表面上。你可能会徒劳地寻找类似下面的函数：

```
DrawBitmap (hdc, hBitmap, xStart, yStart) ; //无此函数！！！
```

该函数将位图拷贝到 hdc 所代表的设备环境上，目标区域从逻辑点（ xStart, yStart ）开始。本章后面，我们将编写自己的 DrawBitmap 函数。不过，你首先必须熟悉几个概念，先从内存设备环境开始。

内存设备环境是这样一种设备环境，它的“显示表面”仅存在于内存中。你可以使用下面的函数：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

来创建内存设备环境，hdc 句柄是已经存在的有效设备环境的句柄，该函数返回一个存储设备句柄。在创建内存设备环境后，所有属性均被设置为普通的缺省值。对于内存设备环境，你几乎可以做想做的一切事情，包括将属性设置为非缺省值，获取属性的当前设置以及将画笔、画刷和区域选入其中，甚至可以在它上面绘图，但是对这一切它不会立即产生反应，原因如下：

当你刚创建内存设备环境时，它具有一个刚好包含一个单色象素的“显示表面”，这真是一个很小的显示表面（不要用 GetDeviceCaps 来获取这些信息，hdcMem 的 HORZSIZE 、 VERTSIZE 、 BITSPIXEL 和 PLANES 值都将被设置为与原 hdc 相关的值。如果 GetDeviceCaps 真的返回与内存设备环境相关的正确值，也就是它创建时的值，那么 HORZSIZE 、 VERTSIZE 、 BITSPIXEL 和 PLANES 都将返回 1 ）。你需要做的是将设备环境的显示表面变大一些，通过将一个位图选进该设备环境中，就可以做到这一点。

```
SelectObject (hdcMem, hBitmap) ;
```

现在，hdcMem 的显示表面与 hBitmap 所代表的位图具有同样的宽度、高度和色彩结构。在缺省的窗口和视口原点下，内存设备环境的逻辑点（0，0）对应于位图的左上角。

如果位图上有某种图形，则该图形现在是内存设备环境显示表面的一部分。对该位图所做的任何改变（例如，通过使用 SetBitmapBits 给位图设置另一个二进制位数组）都反映在这个显示表面中。简而言之，位图就是内存设备环境的显示表面。

前面我们讨论过不同的创建位图的函数，其中一个是：

```
hBitmap = CreateCompatibleBitmap (hdc, xWidth, yHeight) ;
```

如果 hdc 是用于屏幕或打印机的普通设备环境句柄，则位图的色彩平面数目和象素位数与设备的这些信息相同。但是，如果 hdc 是内存设备环境的句柄（而且还没有位图被选进内存设备环境中），则 CreateCompatibleBitmap 将返回一个单色位图，其宽度和高度分别为 xWidth 个象素和 yHeight 个象素。

位图是一个 GDI 对象。在这一章的前面我们已经看到过如何使用 SelectObject 将画笔、画刷或区域选进设备环境中，稍后，我们将会看到如何用这个函数将字体选进设备环境中。你还可以使用 SelectObject 将这四个 GDI 对象选入内存设备环境中，不过，你不能将位图选入普通的设备环境中，而只能选入内存设备环境中。

当不再需要使用内存设备环境时，你必须删除它：

```
DeleteDC (hdcMem) ;
```

现在，你可能会说，这一切都很好，但是我们还没有解决将位图输出到显示器上的问题。我们所做的只是将它选进了内存设备环境中，现在要做什么呢？我们必须学习将位图从一个设备环境“ blt ”（读作“ blit ”）到另一个设备环境中。

4.10.5 强大的 BLT

计算机图形学涉及将象素输出到显示设备。在前面我们看到了完成这一功能的较精炼的方式，不过，就象素操作的能力而言， Windows 中没有什么方式赶得上 BitBlt 及其“近亲” PatBlt 和 StretchBltd 。 BitBlt(读作“ BitBlit ”)代表“位块传送（ bit-block transfer ）”，它是一个象素移动程序，或者称为光栅 Blaster 程序。“传送”一词并不能完全说明 BitBlt，它不仅仅做传送工作，实际上，它是用 256 种不同类型的光栅操作之一去执行三组象素的逻辑组合。

4.10.5.1 PatBlt函数

PatBlt （图案块传输(“ pattern block transfer ”)）是三种“ blt ”函数中最简单的，它实际上与 BitBlt 和 StretchBlt 非常不同，因为它只使用一种设备环境。不过，从 PatBlt 入手是合适的。

在前面，你已经遇到过叫做绘图方式的设备环境属性，这种属性可以设置为 16 种二值光栅运算(ROP2)码之一。在画线时，绘图方式确定 Windows 对画笔象素和设备环境目标象素所执行的逻辑操作类型。 PatBlt 类似于绘图方式，不过它改变设备环境目标的矩形区域，而不只是线。它对这个矩形内的象素与“图案”执行逻辑操作。“图案”不是什么新东西，它只是画刷的另一个名字而已。对于这个图案， PatBlt 使用当前选进设备环境中的画刷。

PatBlt 的语法为：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, dwROP) ;
```

xDest 、 yDest 、 xWidth 和 yHeight 参数使用逻辑单位，逻辑点（ xDest, yDest ）指定矩形的左上角，矩形宽为 xWidth 个单位，高为 yHeight 个单位（下一节“ Blt 坐标”对这些值有更精确的定义）。这是 PatBlt 所改变的矩形区域。 PatBlt 对画刷和目标设备环境执行的逻辑操作由 dwROP 参数确定，它是一个双字（ 32 位整数） ROP 码，而不是绘图方式所用的 ROP2 码之一。

Windows 有 256 种 ROP 码，它们定义了源显示区域、目标显示区域与图案（即画刷）间所有可能的逻辑组合。通过使用某种“编译器”，视频显示器的设备驱动程序支持所有 256 种光栅操作。这个编译器用 32 位 ROP 码在堆栈上创建一系列机器语言指令，这套指令能完成对显示器象素的逻辑操作，然后执行这些指令。 32 位 ROP 码的高位字是 0 到 255 之间的数，低位字是一个数，它帮助设备驱动程序“编译器”构成逻辑操作的机器码。 256 种 ROP 码中有 15 种有名字。

由于 PatBlt 函数只使用目标设备环境和图案（不使用源设备环境），所以它只接受这 256 个 ROP 码的一个子集，也就是只使用目标设备环境和图案的那 16 个 ROP 码。 PatBlt 支持的 16 种光栅操作如下表所示，你会注意到这个表与 P129 上所示的 ROP2 码很相似。

图案(P):	1	1	0	0	布尔	ROP	
目标(D):	1	0	1	0	操作	代码	名称
结果:	0	0	0	0	0	0x000042	BLACKNESS
	0	0	0	1	~(P | D)	0x0500A9	
	0	0	1	0	~P & D	0x0A0329	
	0	0	1	1	~P	0x0F0001	
	0	1	0	0	P & ~D	0x500325	
	0	1	0	1	~D	0x550009	DSTINVERT
	0	1	1	0	P ^ D	0x5A0049	PATINVERT
	0	1	1	1	~(P & D)	0x5F00E9	
	1	0	0	0	P & D	0xA000C9	
	1	0	0	1	~(P ^ D)	0xA50065	
	1	0	1	0	D	0xAA0029	
	1	0	1	1	~P | D	0xAF0229	
	1	1	0	0	P	0xF00021	PATCOPY
	1	1	0	1	P | ~D	0xF50225	
	1	1	1	0	P | D	0xFA0089	
	1	1	1	1	1	0xFF0062	WHITENESS

对于单色设备环境，值为 1 的位对应于白色象素，值为 0 的位对应于黑色象素。在开始考虑 PatBlt 时，目标和图案为纯黑色或者纯白色是最容易的。例如，如果你调用：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, 0x5F00E9L) ;
```

那么仅当目标点为空，并且 WHITE_BRUSH 被选进设备环境时，从逻辑点（xDest, yDest）开始，宽为 xWidth 个象素，高为 yHeight 个象素的矩形区域将被着色为黑，否则，目标将着色为白。当然，即使在单色设备环境中，目标和画刷也都可以是黑色和白色象素的抖动组合。在这种情况下， Windows 以象素为单位执行逻辑组合，这可能产生奇怪的结果。例如，如果目标已经用 GRAY_BRUSH 着色，并且 GRAY_BRUSH 也是当前被选进设备环境的画刷，则：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, PATINVERT) ;
```

将置目标为纯白或纯黑，究竟哪种色彩则取决于目标的抖动象素与画刷的抖动象素如何组合。

彩色引入了许多复杂性，Windows 根据设备存储的组织方式分别对每个色彩平面或者分别对每个色彩位集执行逻辑操作。

PatBlt 的一些较通常的用法如下。如果想要画一个黑色矩形，你可以调用：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, BLACKNESS) ;
```

如果要画一个白色矩形，则使用：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, WHITENESS) ;
```

函数：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, DSTINVERT) ;
```

总是将矩形的色彩翻转。如果 WHITE_BRUSH 被选进当前设备环境，则函数：

```
PatBlt (hdc, xDest, yDest, xWidth, yHeight, PATINVERT) ;
```

也将对矩形进行翻转。

你可能还记得，FillRect 函数使用画刷来填充矩形区域：

```
FillRect (hdc, &rect, hBrush) ;
```

FillRect 函数与下面的程序段等价：

```
hBrush = SelectObject (hdc, hBrush) ;
PatBlt (hdc, rect.left, rect.top,
             rect.right - rect.left ,
             rect.bottom - rect.top, PATCOPY) ;
SelectObject (hdc, hBrush) ;
```

事实上，Windows 就是用这段程序来执行 FillRect 函数的。在调用

```
InvertRect (hdc, &rect) ;
```

时，Windows 将它变换成如下的函数：

```
PatBlt (hdc, rect.left, rect.top,
             rect.right - rect.left,
             rect.bottom - rect.top, DSTINVERT) ;
```

4.10.5.2 Blt坐标

在介绍 PatBlt 函数的语法时，我说过点（xDest, yDest)指定矩形的左上角，矩形宽为 xWidth 个单位，高为 yHeight 个单位。实际上，尽管 Windows 文档也是这样说的，但是这种陈述并不准确。BitBlt、PatBlt 和 StretchBlt 是 GDI 绘图函数中仅有的三个函数，它们根据矩形某一个角开始的逻辑宽度和逻辑高度来指定逻辑矩形坐标，而所有其它使用矩形边框的 GDI 函数都要求坐标以左上角和右下角的方式给出。对于 MM_TEXT 映射方式，上面对 PatBlt 参数的描述是精确的。对于度量映射方式，却并非如此。如果 xWidth 和 yHeight 使用负值，则点（xDest, yDest）将是矩形的左下角。如果想让（xDest, yDest）作为矩形的左上角，yHeight 参数必须设置为矩形高度的负值。

为了更精确，PatBlt 所着色的矩形具有由 xWidth 的绝对值给出的逻辑宽度和由 yHeight 的绝对值给出的逻辑高度，这两个参数均可以为负值。矩形是由逻辑点（xDest, yDest）和（xDest + xWidth, yDest + yHeight）为顶点定义的，矩形的左上角总是包含在 PatBlt 要改变的区域内，右下角在矩形外面。根据映射方式和 xWidth 及 yHeight 参数的符号，矩形左上角的坐标有四种可能：

```
(xDest, yDest)
```

或者

```
(xDest, yDest+yHeight)
```

或者

```
(xDest+xWidth, yDest)
```

或者

```
(xDest+xWidth, yDest+yHeight)
```

如果你将映射方式设置为 MM_LOENGLISH，并想对客户区左上角面积为一平方英寸的地方使用 PatBlt，那么你可以用：

```
PatBlt (hdc, 0, 0, 100, -100, dwROP) ;
```

或者

```
PatBlt (hdc, 0, -100, 100, 100, dwROP) ;
```

或者

```
PatBlt (hdc, 100, 0, -100, -100, dwROP) ;
```

或者

```
PatBlt (hdc, 100, -100, -100, 100, dwROP) ;
```

要将 PatBlt 的参数设置正确，最简单的方法是将 xDest 和 yDeset 设置为矩形的左上角。如果你的映射方式将 y 轴增长方向定义为向上，则 yHeight 参数的值要使用负数。如果你的映射方式将 x 坐标值增长方向定义为向左（这样做还未所未闻），则 xWidth 参数应该使用负值。

4.10.6 用 BitBlt 传输二进制位

从某种意义上说，BitBlt 是 PatBlt 的超集。PatBlt 完成的所有工作它都能完成，并且它在逻辑运算中引入了第二个设备环境。下面是一般的语法：

```
BitBlt (hdcDest, xDest, yDest, xWidth, yHeight ,
        hdcSrc, xSrc, ySrc, dwROP) ;
```

BitBlt 修改由逻辑点（xDest, yDest）和以逻辑单位给出的 xWidth 和 yHeight 所定义的矩形内的目标设备环境（其句柄为 hdcDest），这些参数定义了一个上一节所描述的矩形。BitBlt 也使用源设备环境（其句柄为 SrcDC）中的矩形，该矩形开始于逻辑点（xSrc, ySrc），宽为 xWidth 个逻辑单位，高为 yHeight 个逻辑单位。

BitBlt 对三个元素执行逻辑组合，这三个元素为：选进目标设备环境中的画刷、源设备环境矩形中的象素和目标设备环境矩形中的象素，结果写到目标设备矩形中。你可以使用 256 个 ROP 码的任何一个作 BitBlt 的 dwROP 参数，15 个有名字的 ROP 码如下表所示。

图案(P):	1	1	1	1	0	0	0	0			
源 (S):	1	1	0	0	1	1	0	0	**布尔**	**ROP**	
目标(D):	1	0	1	0	1	0	1	0	**操作**	**代码**	**名称**
结果:	0	0	0	0	0	0	0	0	0	0x000042	BLACKNESS
	0	0	0	1	0	0	0	1	~(S|D)	0x1100A6	NOTSRCERASE
	0	0	1	1	0	0	1	1	~S	0x330008	NOTSRCCOPY
	0	1	0	0	0	1	0	0	S&~D	0x440328	SRCERASE
	0	1	0	1	0	1	0	1	~D	0x550009	DSTINVERT
	0	1	0	1	1	0	1	0	P^D	0x5A0049	PATINVERT
	0	1	1	0	0	1	1	0	S^D	0x660046	SRCINVERT
	1	0	0	0	1	0	0	0	S&D	0x8800C6	SRCAND
	1	0	1	1	1	0	1	1	~S|D	0xBB0226	MERGEPAINT
	1	1	0	0	0	0	0	0	P&S	0xC000CA	MERGECOPY
	1	1	0	0	1	1	0	0	S	0xCC0020	SRCCOPY
	1	1	1	0	1	1	1	0	S|D	0xEE0086	SRCPAINT
	1	1	1	1	0	0	0	0	P	0xF00021	PATCOPY
	1	1	1	1	1	0	1	1	P|~S|D	0xFB0A09	PATPAINT
	1	1	1	1	1	1	1	1	1	0xFF0062	WHITENESS

看看作为逻辑组合结果的 8 个 0 和 1，对应于这些位的两位数字的十六进制数是 ROP 码的高位字。如果我们可以创建一个希望从图案、源和目标得到结果的表，那么我们就能从 Microsoft Developer Studio 的 Reference 部分的 ROP 码表中很容易地确定 ROP 码。后面我们会这样做。如果使用前面表格中所示的 ROP 码之一，那么你可以使用 PatBlt 来代替 BitBlt，因为不必引用源设备环境。

你可以将 hdcSrc 和 hdcDest 设置为相同的设备环境句柄，在这种情况下， BitBlt 将对目标矩形、源矩形和选进设备环境的当前画刷进行逻辑组合。但是，在客户区设备环境中，这样做有点冒险：如果源矩形的某部分被另一个窗口覆盖了，那么 Windows 将以这个窗口的象素为源。 Windows 不知道在你的客户区中这个窗口下面是什么。

不过，源和目标使用相同设备环境的 BitBlt 函数的例子最容易理解。函数：

```
BitBlt (hdc, 100, 0, 50, 100, hdc, 0, 0, SRCCOPY) ;
```

将从逻辑点（0， 0）开始将宽度为 50 个逻辑单位，高度为 100 个逻辑单位的矩形拷贝到从逻辑点（100， 0）开始的矩形区域中。

4.10.7 DrawBitmap 函数

在对已经选进内存设备环境的位图进行操作时， BitBlt 函数是很有价值的。在执行从内存设备环境到客户区设备环境的“位块传输”时，选进内存设备环境的位图被传输给客户区。

前面我们提到过一个虚设的 DrawBitmap 函数，它能在显示表面上画一个位图。这样一个函数的语法如下：

```
DrawBitmap (hdc, hBitmap, xStart, yStart) ;
```

我曾答应要编写一个 DrawBitmap 函数，下面就是此函数：

```
void DrawBitmap (HDC hdc, HBITMAP hBitmap, int xStart, int yStart)
     {
     BITMAP    bm ;
     HDC       hdcMem ;
     DWORD     dwSize ;
     POINT     ptSize, ptOrg ;

     hdcMem = CreateCompatibleDC (hdc) ;
     SelectObject (hdcMem, hBitmap) ;
     SetMapMode (hdcMem, GetMapMode (hdc)) ;

     GetObject (hBitmap, sizeof (BITMAP), (LPVOID) &bm) ;
     ptSize.x = bm.bmWidth ;
     ptSize.y = bm.bmHeight ;
     DPtoLP (hdc, &ptSize, 1) ;

     ptOrg.x = 0 ;
     ptOrg.y = 0 ;
     DPtoLP (hdcMem, &ptOrg, 1) ;

     BitBlt (hdc, xStart, yStart, ptSize.x, ptSize.y,
             hdcMem, ptOrg.x, ptOrg.y, SRCCOPY) ;

     DeleteDC (hdcMem) ;
     }
```

在这里，我们假定你不希望以任何方式放大或者缩小位图，也就是说，如果位图宽为 100 个象素，那么不管映射方式如何，你都希望它占据客户区的 100 个象素宽的矩形。

DrawBitmap 首先使用 CreateCompatibleDC 创建一个内存设备环境，然后使用 SelectObject 将位图选进设备环境，内存设备环境的映射方式与视频设备环境的相同。由于 BitBlt 使用逻辑坐标和逻辑大小，而你又不希望位图放大或缩小，所以，BitBlt 的 xWidth 和 yHeight 参数必须是与位图的物理象素大小对应的逻辑单位。为此，DrawBitmap 用 GetObject 获取位图的尺寸，并用宽和高的值形成一个 POINT 结构，然后，它将这个点转换为逻辑坐标。对位图的原点（设备坐标中的点（0，0））也作类似的处理。

注意，当前选进目标设备环境（hdc）的画刷是无关紧要，因为 SRCCOPY 不使用画刷。

4.10.8 使用各种的 ROP 码

SRCCOPY 确实是 BitBlt 最常用的 dwROP 参数。要完全掌握其它 255 种 ROP 码的使用是很困难的，所以，本节给出了两个例子，这两个例子显示了其它 ROP 码的特征。

在第一个例子中，你有一个单色位图，并想将它输出到屏幕上。但是，我们不想因为显示位图而使黑色（0）位影响当前客户区上的显示，而且，我们想要所有白色（1）位用画刷着色客户区，画刷也许是用 CreateSolidBrush 创建的彩色画刷。那么，怎么做呢？

假定你工作于 MM_TEXT 映射方式，想要从客户区中点(xStart, yStart)开始输出位图。我们已经有了一个单色位图句柄(hBitmap)和彩色画刷句柄(hBrush)，我们还知道位图的宽度和高度，并将它们存放在一个名为 bm 的 BITMAP 结构中。下面就是这些代码：

```
hdcMem = CreateCompatibleDC (hdc) ;
SelectObject (hdcMem, hBitmap) ;
hBrush = SelectObject (hdc, hBrush) ;

BitBlt (hdc, xStart, yStart, bm.bmWidth, bm.bmHeight ,
        hdcMem, 0, 0, 0xE20746L) ;

SelectObject (hdc, hBrush) ;
DeleteDC (hdcMem) ;
```

BitBlt 对目标设备环境（hdc）、源设备环境（hdcMem）和当前选进目标设备环境的画刷执行逻辑组合。所以我们创建一个内存设备环境，将位图选进内存设备环境，将彩色画刷选进客户区显示环境，然后调用 BitBlt。接着，我们再将原画刷选进显示设备环境，并删除内存设备环境。

这段代码唯一令人迷惑的地方是 ROP 代码 0xE20746，这个 ROP 码导致 Windows 执行以下逻辑操作：

```
((Destination ^ Pattern) & Source) ^ Destination
```

还不明显吗？试一试这种方法：

图案:	1	1	1	1	0	0	0	0
源:	1	1	0	0	1	1	0	0
目标:	1	0	1	0	1	0	1	0
结果:	?	?	?	?	?	?	?	?

对于位图（将被选进源内存设备环境）中的每一个黑色位，我们不想改变目标设备环境，这意味着凡是源为 0 的地方，我们就希望结果位与目标相同：

图案:	1	1	1	1	0	0	0	0
源:	1	1	0	0	1	1	0	0

目标:	1	0	1	0	1	0	1	0
结果:	?	?	1	0	?	?	1	0

现在完成了一半。对于位图中每个白色位，我们想用图案来着色目标设备环境。选进目标设备环境的画刷就是这个图案。所以凡是源为1的地方，我们都希望结果等于图案：

图案:	1	1	1	1	0	0	0	0
源 :	1	1	0	0	1	1	0	0
目标:	1	0	1	0	1	0	1	0
结果:	1	1	1	0	0	0	1	0

这意味着ROP码的高位字为0xE2。我们可以在Microsoft Developer Studio中的ROP表中查到它的完整ROP码为0xE20746。

也许现在你会发现当你在创建位图时，把白色位和黑色位搞反了。这很容易改过来，它只不过是另一种逻辑操作而已：

图案:	1	1	1	1	0	0	0	0
源 :	1	1	0	0	1	1	0	0
目标:	1	0	1	0	1	0	1	0
结果:	1	0	1	1	1	0	0	0

现在，ROP码的高位字为0xB8，整个ROP码为0xB8074A，它执行以下逻辑操作：

```
((Destination ^ Pattern) & Source) ^ Pattern
```

下面是第二个例子：当我们涉及图标和光标时，你将发现它们是由两个位图组成的。这两个位图的用法允许这些图形在某些点上是透明的或者将其下面的显示表面反转。对于单色图标或者光标，这两个位图编码如下：

位图1:	0	0	1	1
位图2:	0	1	0	1
结果:	黑	白	屏幕	反转屏幕

Windows将位图1选进内存设备环境，并以ROP码SRCAND调用BitBlt来将位图传输到显示器上。这个ROP码执行如下的逻辑操作：

```
Destination & Source
```

在位图1中，位为1则目标不改变，位为0则被置为0。然后Windows将位图2选进设备环境，并以SRCINVERT为参数调用BitBlt。逻辑操作为：

```
Destination ^ Source
```

在位图 2 中，对于所有为 0 的位，目标不改变；对于所有为 1 的位，反转目标。

看一看表的第一和第二列：使用 SRCAND 的位图 1 使所有位变黑，使用 SRCINVERT 的位图 2 通过反转黑色位，将选中的位变为白色。这些操作设定组成图标或者光标的黑色位和白色位。现在看一看表的第三和第四列。使用 SRCAND 的位图 2 不改变显示，使用 SRCINVERT 的位图翻转选择位的色彩。这些操作使图标或者光标透明，或者反转下面的色彩。

在本章后面对 GrayString 函数的描述中，有创造性地使用 ROP 码的另一个例子。

4.10.9 再论内存设备环境

我们已经使用内存设备环境将已有的位图传输到显示器上。你也能用内存设备环境在位图的表面上绘图，我们将在第 10 章的 GRAFMENU 程序这样做。首先，创建一个内存设备环境：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

然后，创建所期望大小的位图。如果你想创建一个单色位图，那么可以使之与 hdcMem 兼容：

```
hBitmap = CreateCompatibleBitmap (hdcMem, xWidth, yHeight) ;
```

如果要使位图与视频显示器具有相同的色彩结构，那么可以使位图与 hdc 兼容：

```
hBitmap = CreateCompatibleBitmap (hdc, xWidth, yHeight) ;
```

接着，你可以将位图选进内存设备环境：

```
SelectObject (hdcMem, hBitmap) ;
```

现在，你可以使用在这一章中讨论的所有 GDI 函数在这个内存设备环境（更广泛地说，在位图上）画图。当你首次创建位图时，它包含随机位，所以你可能想用以 WHITENESS 或 BLACKNESS 之一的 ROP 码为参数的 PatBlt 函数来清除内存设备环境的背景。

当你在内存设备环境上画完图之后，需要删除它：

```
DeleteDC (hdcMem) ;
```

现在，你有了一个位图，它包含在被选进内存设备环境时所画的一切。

图 4-26 中所示的 SCRAMBLE 程序非常粗糙，也许我根本不应该把它给你看。不过，此程序用一个内存设备环境作为 BitBlt 操作的临时缓存空间，用于交换显示器上两个矩形中的内容。

SCRAMBLE.MAK

```
#------------------------
# SCRAMBLE.MAK make file
#------------------------

scramble.exe : scramble.obj
```

```
    $(LINKER) $(GUIFLAGS) -OUT:scramble.EXE scramble.obj $(GUILIBS)

scramble.obj : scramble.c
    $(CC) $(CFLAGS) scramble.c
```

SCRAMBLE.C

```
/*------------------------------------------------
   SCRAMBLE.C -- Scramble (and Unscramble) Screen
                 (c) Charles Petzold, 1996
  ------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>

#define NUM 200

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static int iKeep [NUM][4] ;
     HDC        hdc, hdcMem ;
     int        cx, cy ;
     HBITMAP    hBitmap ;
     int        i, j, x1, y1, x2, y2 ;

     if (LockWindowUpdate (GetDesktopWindow ()))
          {
          hdc     = CreateDC ("DISPLAY", NULL, NULL, NULL) ;
          hdcMem  = CreateCompatibleDC (hdc) ;
          cx  = GetSystemMetrics (SM_CXSCREEN) / 10 ;
          cy  = GetSystemMetrics (SM_CYSCREEN) / 10 ;
          hBitmap = CreateCompatibleBitmap (hdc, cx, cy) ;

          SelectObject (hdcMem, hBitmap) ;

          srand ((int) GetCurrentTime ()) ;

          for (i = 0 ; i < 2 ; i++)
               for (j = 0 ; j < NUM ; j++)
                    {
                    if (i == 0)
                         {
                         iKeep [j] [0] = x1 = cx * (rand () % 10) ;
                         iKeep [j] [1] = y1 = cy * (rand () % 10) ;
                         iKeep [j] [2] = x2 = cx * (rand () % 10) ;
                         iKeep [j] [3] = y2 = cy * (rand () % 10) ;
                         }
                    else
                         {
                         x1 = iKeep [NUM - 1 - j] [0] ;
                         y1 = iKeep [NUM - 1 - j] [1] ;
                         x2 = iKeep [NUM - 1 - j] [2] ;
```

```
                    y2 = iKeep [NUM - 1 - j] [3] ;
                    }
               BitBlt (hdcMem, 0, 0, cx, cy, hdc,  x1, y1, SRCCOPY) ;
               BitBlt (hdc,  x1, y1, cx, cy, hdc,  x2, y2, SRCCOPY) ;
               BitBlt (hdc,  x2, y2, cx, cy, hdcMem, 0, 0, SRCCOPY) ;

               Sleep (10) ;
               }

          DeleteDC (hdcMem) ;
          DeleteDC (hdc) ;
          DeleteObject (hBitmap) ;

          LockWindowUpdate (NULL) ;
          }

     return FALSE ;
     }
```

图 4-26 SCRAMBLE程序

SCRAMBLE 没有窗口过程。在 WinMain 中，它为整个屏幕获取一个设备环境：

```
hdc = CreateDC ("DISPLATY", NULL, NULL, NULL) ;
```

以及一个内存设备环境：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

然后程序确定整个屏幕的大小并除以 10：

```
xSize = GetSystemMetrics (SM_CXSCREEN)/10 ;
ySize = GetSystemMetrics (SM_CYSCREEN)/10 ;
```

并使用这些大小值来创建一个位图：

```
hBitmap = CreateCompatibleBitmap (hdc, xSize, ySize) ;
```

然后将此位图选入内存设备环境中：

```
SelectObject (hdcMem, hBitmap) ;
```

使用 C 的 rand 函数，SCRAMBLE 产生四个 xSize 和 ySize 整数倍的随机数：

```
x1 = xSize * (rand ()%10) ;
y1 = ySize * (rand ()%10) ;
x2 = xSize * (rand ()%10) ;
y2 = ySize * (rand ()%10) ;
```

程序使用三个 BitBlt 函数交换显示器上的两个矩形。第一个 BitBlt 将从点（x1，y1）开始的矩形拷贝到内存设备环境中：

```
BitBlt (hdcMem, 0, 0, xSize, ySize, hdc, x1, y1, SRCCOPY) ;
```

第二个 BitBlt 将从点（x2, y2）开始的矩形拷贝到从点（x1，y1）开始的位置：

```
BitBlt (hdc, x1, y1, xSize, ySize, hdc, x2, y2, SRCCOPY) ;
```

第三个 BitBlt 将内存设备环境中的矩形拷贝到从点（x2, y2）开始的区域：

```
BitBlt (hdc, x2, y2, xSize, ySize, hdcMem, 0, 0, SRCCOPY) ;
```

整个过程很有效地交换了显示器上两个矩形区域的内容。SCRAMBLE 这样作了 200 次之后屏幕将完全混乱，不过，不用怕，因为 SCRAMBLE 保存了造成这些混乱的过程，在退出之前会使屏幕返回原样。

你也可以用内存设备环境将一个位图的内容拷贝给另一个。例如，假设你想创建一个位图，该位图包含另一个图的左上部四分之一。如果源位图有句柄 hBitmap，那么你可以将其大小取到一个 BITMAP 类型的结构中：

```
GetObject (hBitmap, sizeof (BITMAP), (LPVOID)&bm) ;
```

并创建一个新的四分之一大小的未初始化位图：

```
hBitmap2 = CreateBitmap (bm.bmWidth/2, bm.bmHeight/2 ,
                         bm.bmPlanes, bm.bmBitsPixel, NULL) ;
```

现在，创建两个内存设备环境，并分别将原位图和新创建的位图选进两个内存设备环境中：

```
hdcMem1 = CreateCompatibleDC (hdc) ;
hdcMem2 = CreateCompatibleDC (hdc) ;

SelectObject (hdcMem1, hBitmap) ;
SelectObject (hdcMem2, hBitmap2) ;
```

最后，将头一个位图的左上部四分之一拷贝给第二个位图：

```
BitBlt (hdcMem2, 0, 0, bm.bmWidth/2, bm.bmHeight/2 ,
        hdcMem1, 0, 0, SRCCOPY) ;
```

你做完后，应该将它们清除掉：

```
DeleteDC (hdcMem1) ;
DeleteDC (hdcMem2) ;
DeleteObject (hBitmap) ;
```

4.10.10 色彩转换

如果在 BitBlt 调用中目标和源设备环境的色彩特性不同，Windows 必须将位图从一种色彩格式转换为另一种。当源位图为单色时的色彩转换效果最好，Windows 用目标设备环境的文本色属性来进行此转换：

单色 DC（源）	色彩 DC（目标）
0（黑）	文本色（缺省为黑）
1（白）	背景色（缺省为白）

背景色属性也是 Windows 用来填充点线或者短划线之间的空隙以及影线画刷中影线之间区域的颜色。你可以使用 SetBkColor 来改变背景色。文本色将在这一章后面遇到，它

将确定文本的颜色。你可以用 SetTextColor 来改变文本色。在缺省设置下，单色位图转换成彩色设备环境上的黑白位图。

将彩色设备环境中的源位图转换为单色设备环境中的目标位图的效果却不那么令人满意：

彩色 DC （源）	单色 DC （目标）
象素 != 背景色	0 （黑）
象素 == 背景色	1 （白）

在这种情况下， Windows 使用源设备环境的背景色来确定将哪个色彩转换为白色，而所有其它色彩都转换为黑色。

下面是另一个与色彩有关的问题。 Windows 需要将位图中某种特定的颜色组合（在不同平面上的或者在同一平面上的）作为背景色的 24 位色彩值，这意味着彩色设备环境必须代表一个真实设备或者基于真实设备的内存设备环境。例如，假设你有一个单色设备驱动程序，你在屏幕设备环境的基础上创建了一个内存设备环境，并且选择一个彩色位图到该内存设备环境中，现在，你试着将该位图传送给一个单色设备环境。这是不能成功的，因为Windows不知道内存设备环境位图中每象素的多个色彩平面或者多个位与真实色彩的关系。

4.10.11 映射方式转换

BitBlt 调用的源和目标设备环境需要不同的开始坐标，但是只需要同一个宽度和同一个高度：

```
BitBlt (hdcDest, xDest, yDest, xWidth, yHeight,
        hdcSrc, xSrc, ySrc, dwROP) ;
```

xWidth 和 yHeight 的值使用逻辑单位，它们既适用于源设备环境的矩形，又适用于目标设备环境的矩形。在调用驱动程序文件执行实际操作之前， BitBlt 必须将所有坐标和大小值转换成设备坐标。由于源设备环境和目标设备环境都使用 xWidth 和 yHeight ，它们必须分别转换为每个设备环境的设备坐标（象素）。

如果源设备环境与目标设备环境相同，或者两个设备环境都使用 WM_TEXT 映射方式，则以设备单位计算的矩形尺寸在两个设备环境中将是相同的。这样， Windows 可以作简单的象素对象素传送。但是，如果按设备单位计算，矩形的尺寸在两个设备环境中不同，则 Windows 将工作交给更能干的 StretchBit 函数。

4.10.12 用 StretchBlt 放缩位图

StretchBlt 调用比 BitBlt 调用多了两个参数：

```
StretchBlt (hdcDest, xDest, yDest, xDestWidth, yDestHeight ,
```

```
hdcSrc, xSrc, ySrc, xSrcWidth, ySrcHeight, dwROP) ;
```

由于 StretchBlt 的源矩形和目标矩形接受不同的宽度和高度参数，所以它允许将源设备环境中的位图放大或缩小，以适合目标设备环境中较大或较小的区域。

与 BitBlt 功能是 PatBlt 功能的超集一样， StretchBlt 与 BitBlt 有相同的关系。它允许用户分别指定源矩形和目标矩形的宽度和高度。与 PatBlt 和 BitBlt 一样， StretchBlt 中所有坐标值均使用逻辑单位。 StretchBlt 还允许你在垂直或水平方向上倒转一个图象。如果 xSrcWidth 和 xDestWidth 的符号（当它们被转换为设备单位时）不同，那么 StretchBlt 创建一个镜像，左变成右，右变成左。如果 ySrcHeight 和 yDestHeight 不同，那么 StretchBlt 将图象上下颠倒。

StretchBlt 可能很慢，特别是在对大位图进行操作时。 StretchBlt 还有一些来自缩放位图的问题和困难。在放大位图时， StretchBlt 必须重复象素的行或列，如果放大倍数不是整数，那么该过程将导致图象的某种失真。

当缩小一个位图时， StretchBlt 必须将两个或者多个行或者列象素合并为一行或一列。有三种方法完成此功能，使用哪种方法取决于设备环境中的放缩方式或属性，你可以使用 SetStretchBltMode 函数来改变这个属性：

```
SetStretchBltMod (hdc, iMode) ;
```

iMode 的值可以是下面三个值之一：

- BLACKONWHITE（缺省设置）——如果必须将两个或多个象素组合成一个象素，StretchBlt 对象素执行逻辑“与”操作。仅当所有原象素为白色时，结果象素才为白色，这在实践中意味着黑色象素比白色象素优先。
- WHITEONBLACK ——将两个或者多个象素执行逻辑“或”操作。仅当所有原象素为黑色时，结果象素才为黑色。这在实践中意味着白色象素优先。
- COLORONCOLOR —— StretchBlt 简单除去几行或者几列象素，不进行逻辑组合。这通常是对彩色位图的最好方法，因为用其它两种方式可能导致色彩失真。

4.10.13 画刷和位图

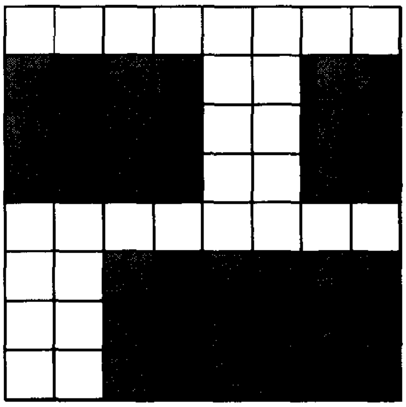
当你使用 CreatePatternBrush 函数或者使用 CreateBrushIndirect 函数并将 lbStyle 域设置为 BS_PATTERN 时，首先需要一个位图句柄。位图必须至少为 8 个象素宽， 8 个象素高，如果更大， Windows 只将位图的左上部分提供给画刷使用。

由于画刷和位图均是 GDI 对象，你必须在程序终止前删除自己创建的画刷和位图。如果基于位图创建画刷，则 Windows 制作一份位图的拷贝，以便在用该画刷绘图时使用。在调用 CreatePatternBrush(或者 CreateBrushIndirect)之后立即删除位图将不会影响画刷，同样，删除画刷也不会影响位图。

我们假设你想画一个矩形，用看似小砖块的画刷填充其内部，如图 4-27 中所示。

图 4-27 以定制画刷填充的图形

你所需要的位图具有如下所示的象素模式：



这是一个单色位图，其高度和宽度均为 8 。

我将在第 9 章中向你演示，你可以将位图创建为一个 Microsoft Developer Studio 的小文件，并将它定义为你程序中的一个资源。装入它将返回一个位图句柄：

```
hBitmap = LoadBitmap (hInstance, "Brick") ;
hBrush = CreatePatternBrush (hBitmap) ;
```

如果你有一个有效的设备环境，那么将这个画刷选进设备环境中，并显示矩形：

```
SelectObject (hdc, hBrush) ;
Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
```

当你释放设备环境之后，删除画刷和位图：

```
DeleteObject (hBrush) ;
DeleteObject (hBitmap) ;
```

你不一定非得等到释放设备环境之后再删除位图，实际上在创建基于位图的画刷之后，任何时候删除位图都是可以的。

你也可以在程序内将位图象素定义为八个无符号整数组成的数组，每个整数对应于位图图案中的一个扫描行，位为 1 代表白色，位为 0 代表黑色：

```
HBITMAP hBitmap ;
HBRUSH hBrush ;
static WORD wBrickBits[]=
     {0xFF, 0x0C, 0x0C, 0x0C, 0xFF, 0xC0, 0xC0, 0xC0} ;
```

位图是用 CreateBitmap 并引用这个整数数组而创建的：

```
hBitmap = CreateBitmap (8, 8, 1, 1,  (LPVOID)wBrickBits) ;
hBrush = CreatePatternBrush (hBitmap) ;
```

接下来的处理与上面类似。

4.11 元 文 件

元文件和矢量图象的关系，就象位图和光栅图象的关系一样。元文件是由人或计算机程序所创建的，由对应于图形函数的一系列记录所组成，例如，绘制直线和曲线、填充区域以及正文。简而言之，元文件是以二进制形式编码的图形函数的集合。

Paint 程序创建位图；而画图程序创建元文件。在一个绘图程序中，你可以很容易地“抓住”某个图形对象（诸如一条直线等）并将它移动到别的位置，这是因为图片的所有单个成员都是以单独的记录而存储的。在画图程序中，这是不可能的——你通常都将局限于删除或者插入位图的矩形块。

由于元文件以图形绘制命令来描述一个图象，因此可以对图象进行缩放，而不会失真；位图却无法提供这种保证。如果你以二倍大小来显示一个位图，你却无法得到二倍的分辨率，而只是在水平和垂直方向上重复位图的位。

可以将元文件转换为位图，但是将丢失一些信息。组成元文件的图形对象将不再是独立的了，而被合并进一个大的图象。将位图转换为元文件是一项更困难的工作，通常只限于很简单的图象，并要求大量的处理能力来分析边界和轮廓。

元文件大多用于程序通过剪贴板共享图片的情形，虽然它们也可以作为剪贴图片(clip art)保存在磁盘上。由于元文件将图片描述为 GDI 调用的集合，因而它们既比位图占用的空间少得多，又比位图与设备更无关。

我将首先讨论 Windows 1.0 以来所支持（并且仍然为 Windows 95 所支持）的元文件函数，然后我将讨论为 Windows NT 开发并由 Windows 95 所支持的“增强的元文件”。增强的元文件在原来元文件的基础上有一些提高，并且更加适合于磁盘存储。

4.11.1 内存元文件的简单应用

你可以用 CreateMetaFile 来创建元文件设备环境从而创建一个元文件，然后可以用大多数 GDI 绘图函数在这个元文件设备环境上绘图。然而这些 GDI 调用并不会真正画什么，而是被存储在元文件中。当你关闭元文件设备环境时，将返回给你一个元文件的句柄。然后你就可以在一个真正的设备环境上“绘制”这个元文件并执行元文件中的 GDI 函数。

CreateMetaFile 要求一个参数，该参数可以是 NULL 或者一个文件名。如果是 NULL，则将元文件存储于内存中。如果是一个文件名（扩展名习惯上为.WMF —— “ Windows Metafile ”），则将元文件存储在磁盘文件中。

图 4-28 中所示的程序 METAFIILE 显示如何在 WM_CREATE 消息期间创建一个内存元文件并在 WM_PAINT 消息期间显示该图象 100 次。

METAFILE.MAK

```
#-----------------------
# METAFILE.MAK make file
#-----------------------

metafile.exe : metafile.obj
     $(LINKER) $(GUIFLAGS) -OUT:metafile.exe metafile.obj $(GUILIBS)

metafile.obj : metafile.c
     $(CC) $(CFLAGS) metafile.c
```

METAFILE.C

```
/*----------------------------------------------
   METAFILE.C -- Metafile Demonstration Program
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName [] = "Metafile" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Metafile Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
```

```
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static HMETAFILE hmf ;
     static int       cxClient, cyClient ;
     HBRUSH           hBrush ;
     HDC              hdc, hdcMeta ;
     int              x, y ;
     PAINTSTRUCT      ps ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdcMeta = CreateMetaFile (NULL) ;
               hBrush  = CreateSolidBrush (RGB (0, 0, 255)) ;

               Rectangle (hdcMeta, 0, 0, 100, 100) ;
               MoveToEx (hdcMeta,   0,   0, NULL) ;
               LineTo   (hdcMeta, 100, 100) ;
               MoveToEx (hdcMeta,   0, 100, NULL) ;
               LineTo   (hdcMeta, 100,   0) ;

               SelectObject (hdcMeta, hBrush) ;
               Ellipse (hdcMeta, 20, 20, 80, 80) ;

               hmf = CloseMetaFile (hdcMeta) ;

               DeleteObject (hBrush) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               SetMapMode (hdc, MM_ANISOTROPIC) ;
               SetWindowExtEx (hdc, 1000, 1000, NULL) ;
               SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;

               for (x = 0 ; x < 10 ; x++)
                    for (y = 0 ; y < 10 ; y++)
                         {
                         SetWindowOrgEx (hdc, -100 * x, -100 * y, NULL) ;
                         PlayMetaFile (hdc, hmf) ;
                         }

               EndPaint (hwnd, &ps) ;
```

```
            return 0 ;

        case WM_DESTROY :
            DeleteMetaFile (hmf) ;
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 4-28 METAFILE程序

这个程序演示了在使用一个内存元文件时所涉及的四个元文件函数的用法。第一个是 CreateMetaFile，在处理 WM_CREATE 消息期间用 NULL 参数调用该函数，将返回一个元文件设备环境的句柄。然后 METAFILE 用这个元文件 DC 来绘制一个矩形、两条直线和一个蓝色椭圆，这些函数调用以二进制形式存储在元文件中。CloseMetaFile 函数返回元文件的句柄，注意元文件句柄被存储在一个静态变量中，因为它将在以后被使用。

元文件包含 GDI 函数调用的二进制表示，这些函数调用是一个 Rectangle 调用、两个 MoveToEx 调用、一个 SelectObject 调用（指定蓝色画刷）和一个 Ellipse 调用。坐标没有指定任何映射方式或者任何转换，它们只是作为数字被存储在元文件中。

在 WM_PAIT 消息期间，METAFILE 设置一种映射方式并调用 PlayMetaFile 在窗口中绘制对象 100 次。元文件中函数调用的坐标按照目标设备环境的当前变换方式加以解释。在调用 PlayMetaFile 时，事实上你是在重复地调用最初在 WM_CREATE 消息期间创建元文件时，在 CreateMetaFile 和 CloseMetaFile 之间所作的所有调用。

和任何一个 GDI 对象一样，元文件对象也应该在程序终止前删除，这在 WM_DESTROY 消息期间用 DeleteMetaFile 函数来实现。

METAFILE 程序的结果如图 4-29 中所示。

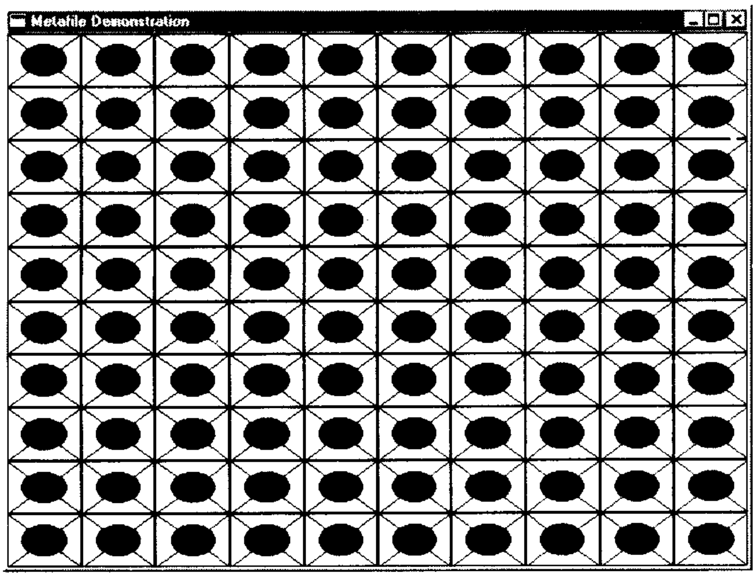


图 4-29 METAFILE显示

4.11.2 将元文件保存在磁盘上

上面的例子中，CreateMetaFile 的 NULL 参数意味着我们想创建存储在内存中的元文件。我们也可以创建与普通文件一样存储在磁盘上的元文件，这种方法对于大的元文件比较合适，因为这样做将使用较小的内存空间，Windows 只需在内存中保持一个相对较小的区域，以存放包含元文件的文件名。而另一方面，将元文件存放在磁盘上就需要在每次运行元文件时进行磁盘访问。

要将METAFILE转换为使用基于磁盘的元文件，你需要将CreateMetaFile的参数NULL替换为一个文件名。在 WM_CREATE 处理结束时，你可以用元文件句柄来调用 DeleteMetaFile，这样，句柄被删除，但是磁盘文件仍然被保存着。

在 WM_PAINT 消息期间，你可以通过调用 GetMetaFile 来获取此磁盘文件的元文件句柄：

```
hmf = GetMetaFile (szFileName) ;
```

现在你可以如前面所示那样显示这个元文件。当 WM_PANT 消息处理结束时，你可以删除这个元文件句柄：

```
DeleteMetaFile (hmf) ;
```

当开始处理 WM_DESTROY 消息时，你不必删除元文件，因为它已经在 WM_CREATE 消息和每一个 WM_PAINT 消息的结束时被删除了，但是你仍然应该删除磁盘文件：

```
remove (szFileName) ;
```

当然，除非你想保存文件。在第 9 章中，我将讨论用户定义的资源，资源通常是存储在一个程序的.EXE 文件中的二进制数据，但是又不同于一般的代码和数据区。元文件可以是这样的一种资源。如果你有一块包含元文件内容的数据，那么你可以使用：

```
hmf = SetMetaFileBitsEx (iSize, pData) ;
```

来创建一个元文件。SetMetaFileBitsEx 有一个对应的函数，GetMetaFileBitsEx，此函数将元文件的内容拷贝到一块存储中。

4.12 增强型元文件

我在上面所描述的老的（但是仍然被支持的）元文件存在一些问题，特别是在一个程序使用另一个程序所创建的元文件时，无法确定元文件所表示图象的大小，必须进入元文件的内部对所有的绘图命令进行检查。这是一个大问题。

早期，Microsoft 建议元文件不要包含对 SetMapMode 或者其它改变窗口/视口变换的函数的调用，这么做可以使元文件与设备相关或者防止应用程序改变显示图象的大小。因此，元文件中的所有坐标都仅仅是一些数字，不与任何一种坐标系相关。

我们将在第 16 章中看到，旧式元文件并不是直接通过剪贴板传递的，剪贴板只是处理一些所谓的“元文件图片”，这是一个 METAFILEPICT 类型的结构。元文件句柄是该结

构的一个域，而映射方式标识符（指示元文件中的 GDI 函数调用所使用坐标的单位）和图象的大小也是其中的域。这些信息有助于程序在输入元文件时，为图象的显示设置适当的 GDI 环境。

元文件图片结构是一种经典的 kludge，把它加到 GDI 中显然是为了弥补元文件格式中的缺陷。而且，没有为磁盘元文件开发类似的支持，这就是你为什么看不到它们广泛用于交换图片的原因。它们太难用了。

4.12.1 改进

与 Windows NT 类似， Windows 95 支持一种新的“增强型元文件”格式，它包括一系列新的函数、几个新的数据结构、一种新的剪贴板格式和一个新的文件扩展名—— EMF 。

最重要的增强是新的元文件格式包括了更多的可以通过一个函数调用来访问的头信息，这些信息能够帮助应用程序显示元文件图象。

一些增强型元文件函数允许你在增强型元文件(EMF)格式与现在叫做 Windows 元文件(WMF)的格式之间进行转换。当然，这样的转换并不是完美的，因为，旧的元文件格式不支持一些新的图形特性（比如路径）。

4.12.2 基本过程

图 4-30 中的 EMF1 程序，创建和显示一个增强型元文件。

EMF1.MAK

```
#--------------------
# EMF1.MAK make file
#--------------------

emf1.exe : emf1.obj
     $(LINKER) $(GUIFLAGS) -OUT:emf1.exe emf1.obj $(GUILIBS)

emf1.obj : emf1.c
     $(CC) $(CFLAGS) emf1.c
```

EMF1.C

```
/*-------------------------------------
   EMF1.C - Enhanced Metafile Demo #1
            (c) Charles Petzold, 1996
  -------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
```

```
     static char szAppName[] = "EMF1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Enhanced Metafile Demo #1",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                LPARAM lParam)
     {
     static HENHMETAFILE hemf ;
     HDC                 hdc, hdcEMF ;
     PAINTSTRUCT         ps ;
     RECT                rect ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdcEMF = CreateEnhMetaFile (NULL, NULL, NULL, NULL) ;

               Rectangle (hdcEMF, 100, 100, 200, 200) ;

               MoveToEx  (hdcEMF, 100, 100, NULL) ;
               LineTo    (hdcEMF, 200, 200) ;

               MoveToEx  (hdcEMF, 200, 100, NULL) ;
               LineTo    (hdcEMF, 100, 200) ;
```

```
               hemf = CloseEnhMetaFile (hdcEMF) ;

               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               rect.left   =     rect.right  / 4 ;
               rect.right  = 3 * rect.right  / 4 ;
               rect.top    =     rect.bottom / 4 ;
               rect.bottom = 3 * rect.bottom / 4 ;

               PlayEnhMetaFile (hdc, hemf, &rect) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               DeleteEnhMetaFile (hemf) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 4-30 EMF1程序

在 EMF1 的 WM_CREATE 消息处理过程中，程序创建了一个增强型元文件。创建过程从一个 CreateEnhMetaFile 开始，这个函数需要四个参数，不过，你可以把它们全设置为 NULL 。（多方便啊！）稍候，我将讨论用非 NULL 参数调用这个函数。

与 CreateMetaFile 一样， CreateEnhMetaFile 返回一个特殊的设备环境句柄。程序使用这个句柄来绘制一个矩形和两条直线，两条直线分别连接矩形的对角顶点。这些函数调用及其参数被转换成二进制形式并存到元文件中。

最后，一个 CloseEnhMetaFile 结束元文件的创建过程并返回一个元文件句柄，这个句柄被保存到一个 HENHMETAFILE 类型的静态变量中。

在 WM_PAINT 消息的处理中， EMF1 将程序客户窗口的大小取到一个 RECT 结构中，该结构的四个域被重新调整，使得矩形的宽和高各是客户窗口的一半，并且处在居中的位置。 EMF1 然后调用 PlayEnhMetaFile ，第一个参数是窗口的设备环境句柄，第二个参数是增强型元文件的句柄，第三个参数是指向那个 RECT 结构的指针。

在创建元文件时， GDI 将计算出整个元文件图象的大小。在这个例子里，图象的宽和高都是 100 个单位。显示这个元文件时， GDI 将放缩图象，以便使它与 PlayEnhMetaFile 中指定的矩形相称。图 4-31 给出了三个 EMF1 的实例在 Windows 95 下运行的情况。

最后，在收到 WM_DESTROY 时， EMF1 通过调用 DeleteEnhMetaFile 来删除元文件。

让我们小结一下在 EMF1 中能学到的东西。

首先，在这个特定的程序中，创建增强型元文件时，在画矩形和直线的函数中使用的坐标并不那么重要。你可以将它们放大两倍，也可以将它们减去一个常数，结果还是一样

的。重要的是在定义一个图象时，在所用的多个坐标之间保证一致性。

其次，传给 PlayEnhMetaFile 的图象被放缩以便使矩形与客户窗口相称。这样，正如图 4-31 所清楚地表明的，图象可以变形。元文件坐标暗示图象是正方形的，但是，这并非我们在一般情况下所得到的结果。

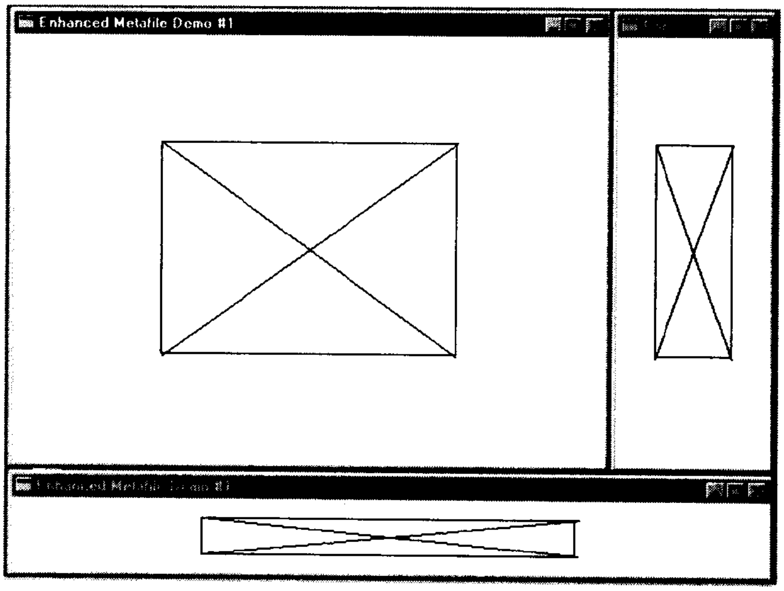


图 4-31 EMF1的显示

有时，这正是你想要的。为了把图象嵌入字处理文本中，你可能会让用户为图象划定一个矩形并使整个图象与矩形精确地吻合在一起而不留任何空白。用户可以适当地调整矩形来得到想要的纵横比。

不过，有时候另一种方式更合适。你可能想要保持原来图象的纵横比，因为它对于准确生成视觉信息可能极为重要。例如，警方的嫌疑犯素描就不应该比原图更胖或者更高。或者，你可能想要保持原来图象的尺寸，也许使图象保持两英寸高会很重要，而不应该去改变它。

4.12.3 内部机制

图 4-32 中的 EMF2 程序创建一个磁盘元文件。

EMF2.MAK

```
#--------------------
# EMF2.MAK make file
#--------------------

emf2.exe : emf2.obj
     $(LINKER) $(GUIFLAGS) -OUT:emf2.exe emf2.obj $(GUILIBS)

emf2.obj : emf2.c
     $(CC) $(CFLAGS) emf2.c
```

EMF2.C

```
/*-----------------------------------------
   EMF2.C - Enhanced Metafile Demo #2
            (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "EMF2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Enhanced Metafile Demo #2",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                  LPARAM lParam)
     {
     HDC          hdc, hdcEMF ;
```

```
HENHMETAFILE hemf ;
PAINTSTRUCT  ps ;
RECT         rect ;

switch (iMsg)
     {
     case WM_CREATE :
          hdcEMF = CreateEnhMetaFile (NULL, "emf2.emf", NULL,
                                      "EMF2\0EMF Demo #2\0") ;

          Rectangle (hdcEMF, 100, 100, 200, 200) ;

          MoveToEx  (hdcEMF, 100, 100, NULL) ;
          LineTo    (hdcEMF, 200, 200) ;

          MoveToEx  (hdcEMF, 200, 100, NULL) ;
          LineTo    (hdcEMF, 100, 200) ;

          hemf = CloseEnhMetaFile (hdcEMF) ;

          DeleteEnhMetaFile (hemf) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;

          rect.left   =     rect.right  / 4 ;
          rect.right  = 3 * rect.right  / 4 ;
          rect.top    =     rect.bottom / 4 ;
          rect.bottom = 3 * rect.bottom / 4 ;

          hemf = GetEnhMetaFile ("emf2.emf") ;

          PlayEnhMetaFile (hdc, hemf, &rect) ;

          DeleteEnhMetaFile (hemf) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 4-32 EMF2程序

注意，在这里 CreateEndMetaFile 的第一个参数是一个设备环境句柄， GDI 使用这个参数将尺寸信息存到元文件头中。如果参数被设为 NULL， GDI 就假定这一尺寸信息是基于视频设备环境的。

第二个参数是文件名。如果你把这个参数设置为 NULL（正如 EMF1 而非 EMF2 所做

的），那么函数将创建一个内存元文件。EMF2 用 EMF2.EMF 创建一个磁盘元文件。

第三个参数是一个 RECT 结构的地址，该 RECT 结构指示整个元文件的大小，这一重要信息（早期 Windows 元文件格式的缺陷所在）被保存在元文件头中。如果你把这个值置为 NULL，GDI 将为你计算出大小来。我喜欢操作系统帮我做事儿，所以我把这个参数置为了 NULL。如果性能对于你的应用程序来讲至关重要，你可能会使用这个参数以免除 GDI 的一些工作。

最后一个参数是描述元文件的一个文本串。这个文本串分为两部分来给出，第一部分是应用程序的名字，以一个 NULL 字符结束；第二部分描述视觉图象，后面跟着两个 NULL 字符。例如，采用 C 的写法，这个描述串可以是"HemiDemiSemiCad V6.4\0FlyingFrogs\0\0"。由于 C 通常将一个 NULL 字符放到引号中的串的末尾，在描述串的最后，你只需要给出一个'\0'就可以，如 EMF2 所做的。

在创建元文件之后，EMF2 的处理与 EMF1 相似，以 CreateEnhMetaFile 返回的设备环境句柄调用几个 GDI 函数，这些操作将绘制一个矩形及其两条对角线，然后，程序调用 CloseEnhMetaFile 来清除设备环境句柄并得到整个元文件的一个句柄。

此后，仍然在 WM_CREATE 处理期间，EMF2 做一些 EMF1 未做的事情：得到元文件句柄后，程序调用 DeleteEnhMetaFile，这将清除用于维护元文件的所有内存资源。然而，磁盘元文件仍然存在（如果你想要删除这个文件，可以使用通常的文件删除函数）。注意，元文件句柄在 EMF2 中没有被保存为静态变量，这就意味着不需要在两条消息的间隔期间保存它。

这样，为了使用元文件，EMF2 就需要访问磁盘文件，它通过在 WM_PAINT 消息处理期间调用 GetEnhMetaFile 来进行。此函数唯一的参数是元文件名，并且函数返回元文件的句柄。EMF2 将这个句柄作为参数调用 PlayEnhMetaFile 函数，这与 EMF1 中一样。这个函数的最后一个参数指定一个矩形，元文件图象就显示在这个矩形中。但是，与 EMF1 不同，EMF2 在结束 WM_PAINT 消息处理时删除元文件。在以后的 WM_PAINT 消息处理中，EMF2 再次取得该元文件，绘制它，并删除它。

记住，这里的删除元文件操作只是删除维护该元文件所需的各种内存资源，磁盘元文件依然存在，直到程序终止以后也是如此。

由于 EMF2 留下一个磁盘元文件，你可以瞧瞧它，它由一些 Windows 头文件里定义的 ENHMETARECORD 结构类型的变长记录构成。增强型元文件总是以一个头记录开始，头记录是 ENHMETAHEADER 结构类型的。

为了获得头记录信息，你无需访问实际的磁盘元文件。如果你有元文件的句柄，你就可以使用 GetEnhMeteFileHeader 函数：

```
GetEnhMetaFile (hemf, cbSize, &emh) ;
```

第一个参数是元文件句柄，最后一个参数是一个 ENHMETAHEADER 结构的指针，第二个参数是这个结构的大小。你可以使用类似的 GetEnhMetaFileDescription 函数来获得描述串。ENHMETAHEADER 的 rclBounds 域是以象素为单位给出图象大小的矩形结构，rclFrame 域是提供同一信息的另一个矩形结构，只不过单位是 0.01 毫米。

头记录以两个 SIZE 结构结束。SIZE 结构含有两个 32 位的域，szlDevice 和

szlMillimeters 。 szlDevice 域以象素为单位给出输出设备的大小；而 szlMillimeters 以毫米为单位给出输出设备的大小。这是由传给 CreateEnhMetaFile 调用的第一个设备环境句柄参数所指定的设备决定的，如果该参数是 NULL ， GDI 将采用视频显示设备的信息。 GDI 通过 GetDeviceCaps 来获得大小信息。

4.12.4 显示精确的图象

元文件的优点在于它可以被放缩到任何尺寸而仍然保持不失真，放大或者缩小图象只需要放缩定义图元的所有坐标点的坐标。与此相反，当压缩导致去掉整行或者整列象素时，位图将丢失至关重要的信息。

不过，有时候任意放缩元文件的想法并不那么有效。尽管放缩一个人脸的元文件让人变的胖些或瘦些可能是挺有趣的，但是知道图象的正确纵横比也会很有益。而且，某些元文件图象只在一种特定的尺寸下才合适（假冒者们或许会喜欢这样的情形）。

我们已经看到， PlayEnhMetaFile 的最后一个参数是一个 RECT 结构，该结构告诉 GDI 你想让图象显示在目标设备环境中的什么地方。 GDI 将放缩图象以便和该矩形精密地吻合。精确地显示一个元文件——不管是按特定的尺寸还是按适当的纵横比——需要使用元文件头中的大小信息并相应地设置矩形结构。

这些操作可能相当复杂，不过让图象在打印机上输出会更有趣些，所以我把这些工作推迟到第 15 章里进行。

4.13 文本和字体

我在这本书中演示的第一个 Windows 程序是在一个窗口的中央显示一行简单的文本串。在上一章中我们能够进一步显示多行并且可以滚动的文本。现在是以一种更严格的方式来检查文本显示技术的时候了。关于文本的讨论不会就此结束，在第 11 章中，我们将看到通用对话框（ Common Dialog Box ，在 Windows 3.1 中引入的）库是如何大大简化那些必须让用户选择字体的程序的。在第 15 章中，我们将探讨有关如何使文本在屏幕上的显示与它最终在纸上的打印效果完全一样的问题。

4.13.1 简单的文本输出

我们来看看 Windows 为文本输出提供的几个函数、影响文本的设备环境属性以及“现有”字体的使用。

最常用的文本输出函数就是我在第三章的 SYSMETS 程序中使用的 TextOut 函数：

```
TextOut (hdc, xStart, yStart, pString, iCount) ;
```

xStart 和 yStart 参数是文本串开始位置的逻辑坐标。通常，这一点是 Windows 开始输出的第一个字符的左上角的位置。 TextOut 还需要一个指向字符串的远指针和串的长度，函数

不识别以 NULL 结束的字符串。

传给 TextOut 的 xStart 和 yStart 参数的含义可以用 SetTextAlign 函数来改变。TA_LEFT 、 TA_RIGHT 和 TA_CENTER 标志影响 xStart 如何在水平方向上定位串，缺省设置为 TA_LEFT 。如果在 SetTextAlign 函数中指定 TA_RIGHT ，则其后的 TextOut 调用将串中最后一个字符的右边定位在 xStart 处。对于 TA_CENTER ，串的中心定位在 xStart 处。

类似地， TA_TOP 、 TA_BOTTOM 和 TA_BASELINE 标志影响纵向定位， TA_TOP 为缺省设置，它意味着串中字符的上边被定位在 yStart 所指定的位置。使用 TA_BOTTOM 意味着串被定位到其基线处于 yStart 的位置。基线是一条线，下垂字母的下部在它下面（如小写的 p ， q ， y ）。

如果你使用 TA_UPDATECP 标志调用 SetTextAlign ， Windows 将忽视 TextOut 的 xStart 和 yStart 参数，而用先前由 MoveTo 或者 LineTo 设置的当前位置代替。 TA_UPDATECP 标志还导致 TextOut 函数更新串尾的当前位置（ TA_LEFT ）或者串头的当前位置（ TA_RIGHT ），这对于使用多个 TextOut 调用来显示一行文本是有用的。当横向定位为 TA_CENTER 时，在调用 TextOut 之后，当前位置不变。

你应该还记得，在第 3 章的一系列 SYSMETS 程序中，分列显示文本需要每列调用一次 TextOut 。另一种选择是使用 TabbedTextOut 函数：

```
TabbedTextOut (hdc, xStart, yStart, pString, iCount ,
               iNumTabs, piTabStops, xTabOrigin) ;
```

如果文本串包含有嵌入的制表字符（ ‘\t’ 或者 0x09 ），则 TabbedTextOut 函数将根据传送给函数的一个整数数组将制表字符展开为空格。

TabbedTextOut 的前五个参数与 TextOut 的参数相同，第六个参数是制表符的停止位数目，第七个参数是制表符停止位数组，单位为象素。例如，如果平均字符宽度为 8 个象素，并且你希望每个制表符占 5 个字符，那么该数组将包含数 40 ， 80 ， 120 等，按递增顺序排列。

如果第六个参数与第七个参数为 0 或者 NULL ，那么制表符宽度设置为平均每个制表符占 8 个字符。如果第六个参数为 1 ，那么第七个参数指向一个整数，这个整数是所有制表符跳过的象素数。例如，如果第六个参数为 1 ，第七个参数指向包含数 30 的变量，那么制表符位置为 30 ， 60 ， 90 …个象素。最后一个参数给出了第一个制表符位置的逻辑 x 坐标，这与串的开始位置可能相同，也可能不相同。

另一个高级文本输出函数是 ExtTextOut （ Ext 前缀表示扩展的）：

```
ExtTextOut (hdc, xStart, yStart, iOptions, &rect, pString, iCount, pxDistance) ;
```

第五个参数是一个指针，指向一个矩形结构，它可以是一个剪裁矩形（如果 iOptions 被设置为 ETO_CLIPPED ），也可以是使用当前背景色填充的背景矩形（如果 iOptions 被设置为 ETO_OPAQUE ）。你可以两个都指定，也可以一个也不指定。

最后一个参数是整数数组，它指定串中连续字符之间的空隙。这使得程序可以压紧或者拉松字符间的距离。有时，为了在一窄列中调整一个文本单词，需要指定此参数。这个

参数可以设置为 NULL，以获得缺省的间隙。

一个更高级的文本输出函数是 DrawText，我们在第 2 章的 HELLOWIN 程序中使用过它。对于 DrawText，不是提供一个坐标起始位置，而是给出一个 RECT 类型的结构，它定义了你想让文本在其中显示的矩形：

```
DrawText (hdc, pString, iCount, &rect, iFormat) ;
```

与其它文本输出函数一样，DrawText 需要一个指向字符串的远指针和串长度。不过，如果你使用以 NULL 结尾的字符串，那么你可以将 iCount 设置为－1，这时 Windows 会计算出串的长度。

当 iFormat 被设置为 0 时，Windows 将文本解释为一系列由回车字符（'\r' 或 0x0D）或者换行字符（'\n' 或 0x0A）分开的文本行。文本从矩形的左上角开始显示。回车或换行被解释为“开始新行”字符，因此 Windows 结束当前行，并开始新的一行。新行从矩形的左边开始，比上一行低一个字符（没有额外的留空）。任意要显示到矩形的右边或者下边的文本，包括字符的一部分在内，均被剪裁掉。

你可以通过包含一个 iFormat 参数来改变 DrawText 的缺省操作。iFormat 包含一个或者多个标志，DT_LEFT 标志（缺省设置）指定文本行左对齐，DT_RIGHT 指定文本行右对齐，DT_CENTER 指定文本行排在矩形左边和右边的中间。由于 DT_LEFT 值为 0，所以如果你想让文本右对齐，那么就不需要包含任何标识符。

如果你不希望回车或换行被解释为开始新行字符，那么你可以包含标识符 DT_SINGLELINE，于是，Windows 将回车和换行解释为可显示字符，而不再是控制字符。在使用 DT_SINGLELINE 时，你还可以指定文本行是放在矩形的顶部（DT_TOP，缺省数组），还是放在矩形的底部（DT_BOTTOM），或者是放在正中间（DT_VCENTER）。

在显示多行文本时，Windows 通常只在回车或者换行时结束一行。但是，如果一行太长，矩形装不下，那么可以使用 DT_WORDBREAK 标志，它将导致 Windows 在行内某字结束处换行。对于单行和多行文本显示来说，Windows 都将截去落在外面的文本。通常你可以通过包括 DT_NOCLIP 标志来改变这种情况，并提高函数的操作速度。当 Windows 排列多个文本行时，使用的通常是字符高度，并且没有额外的间距。如果你希望行中间包含额外的间距，那么你可以使用 DT_EXTERNALLEADING 标志。

如果你的文本中包含有制表符（'\t' 或者 0x09），那么你需要包含标志 DT_EXPANDTABS。按照缺省情况，制表位设置为每八个字符一个。你可以使用标志 DT_TABSTOP 来指定另一种制表符设置，在这种情况下，iFormat 的高位字节包含了每个新制表符的字符位置数。不过，我建议你不要使用 DT_TABSTOP，因为 iFormat 的高位字节也用作其它标志。

4.13.2 文本的设备环境属性

有几种影响文本的设备环境属性。在缺省设备环境下，文本色为黑色，但是你可以使用 SetTextColor 函数来改变它：

```
SetTextColor (hdc, rgbColor) ;
```

与画笔色彩和影线画刷色彩一样， Windows 将 rgbColor 转换为纯色。你可以通过调用 GetTextColor 来获取当前文本的色彩。

字符笔划之间的空隙是根据背景方式和背景色的设置来填充的。你可以使用函数 SetBkMode 来改变背景方式：

```
SetBkMode (hdc, iMode) ;
```

其中， iMode 或者为 OPAQUE ，或者为 TRANSPARENT 。缺省背景方式为 OPAQUE ，这意味着 Windows 使用背景色来填充字符笔划之间的空隙区域。你可以使用函数 SetBColor 来改变背景色：

```
SetBkColor (hdc, rgbColor) ;
```

rgbColor 的值被改变为纯色的值，缺省背景色为白色。如果背景方式设置为 TRANSPARENT ， Windows 将忽略背景色，不给字符笔划之间的区域着色。 Windows 还使用背景方式和背景色来为点式线和短划线间的间隙和影线画刷中的影线之间的区域着色，这一点我在前面已经讨论过了。

许多 Windows 程序都指定 WHITE_BRUSH 为画刷， Windows 使用它来删除窗口的背景。画刷是在窗口类结构中指定的。不过，你可能想使程序窗口的背景与用户在控制板中所设置的系统色彩一致，在这种情况下，你可以在 WNDCLASS 结构中使用下面的语句指定背景色。

```
wndclass.hbrBackgroud = (HBRUSH) (COLOR_WINDOW+1) ;
```

当你想向客户区输出文本时，你可以使用当前系统色彩来设置文本色和背景色：

```
SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT) ;
SetBkColor (hdc, GetSysColor (COLOR_WINDOW) ;
```

在此之后，你希望一旦系统色彩变化，你的程序也跟着变化：

```
case WM_SYSCOLORCHANGE:
     InvalidateRect (hwnd, NULL, TRUE) ;
     break ;
```

另一个影响文本的设备环境属性是字符间距。在缺省情况下，它被设置为 0 ，这时， Windows 在字符间不加入任何空隙。你可以使用函数 SetTextCharacterExtra 插入空隙：

```
SetTextCharacterExtra (hdc, iExtra) ;
```

iExtra 参数使用逻辑单位， Windows 将它转换成最相近的象素数，可以为 0 。如果 iExtra 使用负数值（比如尝试将字符挤得更紧一些）， Windows 将取该数的绝对值；也就是说，不能使该值小于 0 。调用 GetTextCharacterExtra 可以获取当前字符间距，在返回时，

Windows 将象素间距转换为逻辑单位。

4.13.3　使用现有字体

当你调用 TextOut 、 TabbedTextOut 、 ExtTextOut 或者 DrawText 来输出文本时，Windows 使用当前选入设备环境的字体，字体定义了一个特定的字型(typeface)和大小。使用不同字体显示文本的最简单的方法是使用 Windows 提供的现有字体，不过，它们的范围很受限制。

你可以通过调用 GetStockObject 来获取现有字体句柄：

```
hFont = GetStockObject (iFont) ;
```

其中， iFont 是若干个标识符之一，这些标识符中只有两个是常用的。然后，你可以将该字体选入设备环境中：

```
SelectObject (hdc, iFOnt) ;
```

或者你可以将这两条语句合并为一条：

```
SelectObject (hdc, GetStockFont (iFont)) ;
```

GetStockFont 与我们在前面用来获取现有画笔和画刷的函数一样；SelectObject 曾经被我们用来将画笔、画刷、位图和区域选入设备环境中。

选入缺省设备环境的字体叫做系统字体，由 GetStockObject 的 SYSTEM_FONT 参数标识，这是 Windows 在菜单、对话框、消息框和窗口标题条中显示文本时所使用的 ANSI 比例字符集字体。在 GetStockObject 中指定 SYSTEM_FIXED_FONT （我在这一章前面的 WHATSIZE 程序中就是这么做的)能返回固定间距 ANSI 字体的句柄，这种字体与 Windows 3 之前版本中使用的系统字体兼容。当所有的字体字符都具有相同的宽度时，这非常方便。

当你将一种新的字体选入一个设备环境中时，你必须使用 GetTextMetrics 计算出字符的高度和平均宽度。如果你选择的是一种比例字体，那么你需要意识到平均宽度只是“平均值”，有些字符更宽，但是有些字符却较窄。在本章后面，你将学到如何使用 GetTextExtent 来计算由可变宽度字符组成的串的总宽度。

尽管 GetStockObject 的确提供了访问不同字符的最简单的方法，但是你对 Windows 提供的字体没有多少的控制能力。你将马上看到如何根据自己需要设置字型和大小的具体细节。

4.13.4　字体类型

Windows 支持的字体分为两大类，一类是“ GDI 字体”，另一类是“设备字体”。GDI 字体存储在硬盘文件中，设备字体特定于输出设备。例如，对于打印机来说，包含内嵌的设备字体集是非常普遍的。

GDI 字体有三种风格——光栅字体、笔划字体和 TrueType 字体。

光栅字体有时被叫做位图字体，因为在一个光栅字体文件中，每个字符都是以一种位图象素模式存储的。每种光栅字体都是为特定纵横比和字符大小设计的。Windows 可以根据 GDI 光栅字体通过简单地重复象素行或者列来创建较大的字符尺寸，不过，这只能放大整数倍，并且在某种限制范围之内。由于这一原因，GDI 光栅字体被称为“不可缩放”的字体，它们不能被放大或缩小至任意尺寸。光栅字体的主要优点是性能（因为它们显示起来很快）和可读性（因为它们是人工设计的，在设计中尽可能保证了它容易辨认）。

在 Windows 3.1 之前，Windows 还支持另外一种 GDI 字体：GDI 笔划字体。笔划字体以“将点连起来”的格式定义为一系列线段。笔划字体可以随意缩放，这意味着同一字体可以用于任意分辨率的尺寸。但是，性能很差，在尺寸太小时很不容易辨认；尺寸太大时，字符又显得很单薄，因为笔划只是一条线。现在，笔划字体有时被称为“绘图仪字体”，因为它们对绘图仪特别适合，而对其它设备却不怎么适合。

对于光栅字体和 GDI 笔划字体，Windows 都可以“合成”它的黑体、斜体、下划线和刺穿体，不用为每种属性专门存储一种单独的字体。例如，对于斜体，Windows 只是将字符的上面部分向右移一移。

接下来是 TrueType 字体，这一章的剩余部分都将专门讨论这种字体。

4.13.5 TrueType 字体

在 Windows 3.1 中，TrueType 的引入大大增强了使用文本的灵活性。TrueType 是由 Apple 公司和 Microsoft 公司共同开发的一种轮廓字体技术；并且被许多字体供应商所支持。TrueType 字体使用直线和曲线定义每个字符的轮廓，这样，Windows 可以通过改变定义轮廓的坐标，对这些字体进行缩放。TrueType 字体既可以用于视频显示器上，也可以用于打印机上，从而实现了真正的 WYSIWYG（“所见即所得”）。

当你的程序想要使用特定大小的 TrueType 字体时，Windows 将字体“光栅化”，这意味着 Windows 使用 TrueType 字体文件中包含的“提示(hints)”来缩放与每个字符的直线和曲线相关联的坐标。这些“提示”补偿舍入误差，否则可能引起字符无法看清的后果。（例如，在有些字体中，大写 H 的两条腿必须一样宽，对字体盲目缩放可能导致一条腿比另一条腿宽一个象素。提示将能防止这种情况的发生。）每个字符的结果轮廓用于生成字符的位图，这些位图被暂存在存储中，以备将来的使用。

Windows 95 内包含有 13 种 TrueType 字体，它们的“字型名”如下所示：

Courier New
Courier New Bold
Courier New Italic
Courier New Bold Italic
Times New Roman
Times New Roman Bold
Times New Roman Italic
Times New Roman Bold Italic
Arial
Arial Bold
Arial Italic

Arial Bold Italic
Symbol

Courier New 是一种固定间距的字体（也就是说，每个字符具有相同的宽度），它被设计成看起来像是叫做打字机的那种古老设备的输出。Times New Roman 是最初为 Times of London 设计的用于许多打印材料的 Times 字体的一种发展，它被认为具有高度的可读性。Arial 是 Helvetica 演化来的，它是一种无衬线字体，也就是说笔划结尾处没有那个小弯。Symbol 字体由一些常用符号组成。

在传统的排版中，通过字型名和大小来给出一种字体。大小用一种叫做“点”的单位来表达，一个“点”非常接近于 1/72 英寸，因此在计算机排版中经常就把它定义为 1/72 英寸。有的时候，大小被认为是字符上出部分(ascender)的顶端到其下垂部分(descender)的底端之间的距离。这是考虑字体大小时的一种方便的做法，但是这并非对于所有字体都是精确的。有时候，字体设计者会采用一些特别的设计。

4.13.6　EZFONT 系统

TrueType 字体的引入以及它在传统排版中的渊源为 Windows 将文本显示成各种样子提供了坚实的基础。不过，Windows 的某些字体选择函数基于更老的技术，这种技术用屏幕上的光栅字体来近似打印机设备字体。

在本书以前的版本中，我详细地讨论了 EnumFonts 函数，这个函数向程序返回用于 Windows 的每种 GDI 字体和所有设备字体的 LOGFONT（逻辑字体）和 TEXTMETRIC 结构。ChooseFont 对话框（将在第十一章讨论）基本上使 EnumFonts 及其附属的东西不再必要。我也详细讨论过 CreateFontIndirect 函数，它允许程序描述一种想要的字体而不用直接命名它，这就是在视频显示上逼近打印机字体的方法。

与过去不同，这次我要讨论另外一些东西。我想向你表明，如何以最小的工作量和与传统出版相一致的方式在你的程序中使用各种 TrueType 字体。这需要指定字体的名称（也就是前面列出的十三个名字之一）及其大小（我很快会讨论到）。我把这样的程序叫做 EZFONT（“easy font”），图 4-33 给出了你所需要的两个文件。

EZFONT.H

```
/*----------------------
   EZFONT.H header file
  ----------------------*/

HFONT EzCreateFont (HDC hdc, char * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes) ;

#define EZ_ATTR_BOLD          1
#define EZ_ATTR_ITALIC        2
#define EZ_ATTR_UNDERLINE     4
#define EZ_ATTR_STRIKEOUT     8
```

EZFONT.C

```
/*------------------------------------
   EZFONT.C - Easy Font Creation
             (c) Charles Petzold, 1996
 ------------------------------------*/

#include <windows.h>
#include <string.h>
#include <math.h>
#include "ezfont.h"

HFONT EzCreateFont (HDC hdc, char * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes)
     {
     FLOAT      cxDpi, cyDpi ;
     HFONT      hFont ;
     LOGFONT    lf ;
     POINT      pt ;
     TEXTMETRIC tm ;

     SaveDC (hdc) ;

     SetGraphicsMode (hdc, GM_ADVANCED) ;
     ModifyWorldTransform (hdc, NULL, MWT_IDENTITY) ;
     SetViewportOrgEx (hdc, 0, 0, NULL) ;
     SetWindowOrgEx   (hdc, 0, 0, NULL) ;

     if (fLogRes)
          {
          cxDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSX) ;
          cyDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSY) ;
          }
     else
          {
          cxDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, HORZRES) /
                                  GetDeviceCaps (hdc, HORZSIZE)) ;

          cyDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, VERTRES) /
                                  GetDeviceCaps (hdc, VERTSIZE)) ;
          }
     pt.x = (int) (iDeciPtWidth  * cxDpi / 72) ;
     pt.y = (int) (iDeciPtHeight * cyDpi / 72) ;

     DPtoLP (hdc, &pt, 1) ;

     lf.lfHeight         = - (int) (fabs (pt.y) / 10.0 + 0.5) ;
     lf.lfWidth          = 0 ;
     lf.lfEscapement     = 0 ;
     lf.lfOrientation    = 0 ;
     lf.lfWeight         = iAttributes & EZ_ATTR_BOLD      ? 700 : 0 ;
     lf.lfItalic         = iAttributes & EZ_ATTR_ITALIC    ?   1 : 0 ;
     lf.lfUnderline      = iAttributes & EZ_ATTR_UNDERLINE ?   1 : 0 ;
     lf.lfStrikeOut      = iAttributes & EZ_ATTR_STRIKEOUT ?   1 : 0 ;
     lf.lfCharSet        = 0 ;
     lf.lfOutPrecision   = 0 ;
     lf.lfClipPrecision  = 0 ;
```

```
     lf.lfQuality        = 0 ;
     lf.lfPitchAndFamily = 0 ;

     strcpy (lf.lfFaceName, szFaceName) ;

     hFont = CreateFontIndirect (&lf) ;

     if (iDeciPtWidth != 0)
          {
          hFont = (HFONT) SelectObject (hdc, hFont) ;

          GetTextMetrics (hdc, &tm) ;

          DeleteObject (SelectObject (hdc, hFont)) ;

          lf.lfWidth = (int) (tm.tmAveCharWidth *
                              fabs (pt.x) / fabs (pt.y) + 0.5) ;

          hFont = CreateFontIndirect (&lf) ;
          }

     RestoreDC (hdc, -1) ;

     return hFont ;
     }
```

图 4-33 EZFIBT文件

EZFONT.C 只有一个函数，叫做 EzCreateFont，你可以如下使用它：

```
hFont = EzCreateFont (hdc, szFaceName, iDeciPtHeight,
                      iDeciPtWidth, iAttribute, fLogRes) ;
```

此函数返回一个字体句柄，该字体可以通过调用 SelectObject 来选入设备环境中。然后，你应该调用 GetTextMetrics 或 GetOutlineTextMetrics 来确定逻辑坐标下字体的实际大小。在你的程序终止前，应该调用 DeleteObject 来删除任何已经创建的字体。

szFaceName 参数是前面列出的 13 种 TrueType 字型名之一。如果你的系统中安装了其它 TrueType 字体，你也可以使用它们的名字，只是前面列出的那 13 种字体肯定可以为所有 Windows 95 系统所支持。

第三个参数指示想要的字体大小，不过，它是以“十分点”（一个点的十分之一）为单位给出的。这样，如果你想要 12.5 个点的大小，就应该使用 125 。

通常情况下，第四个参数被设置为 0 或者与第三个参数相同。不过，你也可以把这个参数设置为其它值来创建一种宽一点或者窄一点的 TrueType 字体(这有时叫做字体的“ M 宽度”，它描述字体的以点为单位的宽度，不要把它与字体字符的平均宽度什么的相混淆。在排版的早期，大写“ M ”的宽度和高度是一样的，所以才有了“ M 正方形”这种东西，这也就是“ M 宽度”的来历。当“ M 宽度”等于“ M 高度”（即字体的以点为单位的大小）时，字符宽度就是字体设计者所期望的。大一些或者小一些的“ M 宽度”允许你创建宽一点或者窄一点的字符）。

iAttributes 参数可以设为下列定义在 EZFONT.H 中的值的一个或多个：

EZ_ATTR_BOLD
EZ_ATTR_ITALIC
EZ_ATTR_UNDERLINE
EZ_ATTR_STRIKEOUT

通常，你不会用到 EZ_ATTR_BOLD 或者 EZ_ATTR_ITALIC ，因为这些属性是完整的 TrueType 字型名的一部分。如果你使用了它们， Windows 将合成这些效果。

最后，你把最后一个参数设为 TRUE 以使得可见字体的大小根据 GetDeviceCaps 函数返回的“逻辑分辨率”来计算，否则，就根据实际分辨率来计算。

4.13.7 内部机制

EzCreateFont 函数是设计用于 Windows 95 或者 Windows NT 的。SetGraphicsMode 和 ModifyWorldTransform 函数的使用表明了这种 NT 兼容性，这两个函数在 Windows 95 下不起作用（简而言之， Windows NT 的世界变换会影响字体的可见大小，因此在计算字体大小前，它被设置成缺省值——不变换）。

总的说来， EzCreateFont 的功能就是设置一个 LOGFONT 结构的各个域，并调用 CreateFontIndirect ，它会返回字体的句柄。选择 TrueType 字体时，大多数域都设置为 0 。需要你设置的域包括：

- lfHeight ——这是所期望的以逻辑单位计算的字符高度（包括内部留空(leading)而不包括外部留空）。由于字体本身以点计算的大小是字体高度减去内部留空，所以，你实际上在这里指定的就是行距。你可以把 lfHeight 设置成 0 ，以便使用缺省值。如果你把 lfHeight 设置成负数， Windows 将把它的绝对值视为所期望的字体高度而不是行距。如果你想要一种特定的点大小，那就必须将其转换成逻辑单位，并将 lfHeight 域设置成所需结果的负值。
- lfWidth ——这是所期望的以逻辑单位计算的字符宽度。在绝大多数情况下，你把它设置成 0 ，以让 Windows 只根据高度来选择字体。
- lfWeight ——把这个域设置成 700 就可以选择粗体。
- lfItalic ——当这个域是非零值时，选择斜体。
- lfUnderline ——当这个域是非零值时，选择下划线字体。
- lfStrikeOut ——当这个域是非零值时，字体的字符中间有一条横线穿过。
- lfFaceName （ BYTE 数组)——这是字型名(比如 Courier New 、 Arial 或者 Times New Roman ）。

EzCreateFont 的一项任务是将点大小转换成逻辑单位，以设置 LOGFONT 结构。实际上，点大小必须先被转换成设备单位（象素），然后才能转换到逻辑单位。为完成第一步，我们利用从 GetDeviceCaps 函数返回的信息。

以 HORZRES 和 VERTRES 参数调用 GetDeviceCaps 函数，就能得到以象素为单位的视频显示器(或者打印页面的可打印区域)的宽和高。以 HORZSIZE 和 VERTSIZE 参数调用 GetDeviceCaps 函数，就能得到以毫米为单位的视频显示器（或者打印页面的可打印区域)的宽和高。如果最后一个参数设置为 FALSE ， EzCreateFont 函数就根据这些值来计算

出设备的DPI（每英寸点数）分辨率。

有另外一种方法可以达到相同目的。你可以用LOGPIXELSX和LOGPIXELSY参数调用GetDeviceCaps函数来直接获得设备的DPI分辨率，这叫做设备的“逻辑分辨率”。对于打印机而言，普通分辨率和逻辑分辨率是相同的（忽略舍入误差）。然而，对于视频显示设备，逻辑分辨率要小于普通分辨率。例如，对于VGA，按前面的方法计算出来的普通分辨率是大约68DPI，而逻辑分辨率是96DPI。

这个差别是巨大的：假设我们在用一种12点的字体，它是六分之一英寸高。如果普通分辨率是68DPI，一个12点高的字符大约有11个象素高，而对于96DPI的逻辑分辨率，应该是16个象素，增加了45个百分点。

为何会有这个差值呢？如果你仔细想想，实际上并没有什么真正的VGA分辨率。标准VGA显示为640象素宽480象素高，而实际屏幕的大小可以是小小的笔记本计算机，也可以是很大的VGA投影仪。HORZSIZE和VERTSIZE值是基于一种所谓“标准”VGA屏幕大小的，这个大小或许是在1987年的某个时候，由某个不知姓名的Microsoft程序员用尺子从IBM的一种早期型号的VGA屏幕上量出来的。

如果你再想想，会发现实际上并不需要按照它们真正的大小在屏幕上显示字体。假设你正在使用一个VGA投影仪向数百人介绍一些信息，而你使用实际在投影屏幕上也只有六分之一英寸的12点字体。如果你的听众感到迷惑不解甚至心怀敌意，我都不会感到奇怪。

在应用程序中使用各种文本的人们，往往是在进行字处理或者桌面出版工作。这些工作的结果经常是要最终输出到8.5英寸乘11英寸（除去页边留白后是8英寸乘11英寸）的纸张上去。许多VGA显示器都比8英寸宽，根据逻辑分辨率来显示更大的字体将能够更好地利用屏幕的实际性能。

然而，如果你使用逻辑分辨率，在图文混排时可能会有问题。如果你使用SetMapMode函数使得绘图以英寸或者毫米进行，而同时使用逻辑设备分辨率来调整字体大小，你就会遇到不一致之处——不是在打印机的输出上（因为，在那里，普通分辨率和逻辑分辨率是一回事），而是在视频显示器上，这里存在着45个百分点的差值。在本章后面部分的JUSTIFY1程序中，我将演示一种解决方案。

传给CreateFontIndirect的LOGFONT结构需要按逻辑单位计算的字体高度。在你得到以象素计算的值之后，似乎只需要一个简单的DPtoLP（“设备点到逻辑点”）调用就可以得到逻辑单位了。但是，为了让DPtoLP转换正确进行，必须保证映射方式与今后显示所创建字体时的映射方式相同，这就意味着，你应该在调用EzCreateFont之前设置你的映射方式。在多数情况下，你只使用一种映射方式在窗口的一个特定区域上绘图，因此，这个要求应该不是什么问题。

4.13.8 简单的文本格式化

有了EZFONT之后，我们可以试试文本格式化了。这一过程涉及将页边之内的每一行文本用四种方式之一进行定位：向左页边对齐、向右页边对齐、居中或者两端对齐(justified)——就是说，左页边与右页边都对齐，并且在单词之间保持相等的间距。对于前三种情况，

你可以采用以 DT_WORDBREAK 参数调用 DrawText 函数的方法。不过，这一方法有其局限。例如，你不能确定 DrawText 能将文本的那一部分绘制到矩形中。对于一些简单任务来说， DrawText 是方便的，但是对于更为复杂的格式化任务，你还是需要使用 TextOut 。

处理文本时最有用的函数之一是 GetTextExtentPoint32 （这个函数的名字反映出早期 Windows 版本以来的一些变化）。这个函数根据当前选入设备环境的字体计算一个字符串的宽度和高度：

```
GetTextExtentPoint32 (hdc, pString, iCount, &size) ;
```

宽度和高度从 SIZE 结构的 cx 和 cy 域返回。我用一行文本作为例子来讨论这个函数，比如，你已经把一种字体选入了设备环境中，现在想输出下面的文本：

```
char *szText [] = "Hello, how are you?" ;
```

你想让文本从垂直方向的 yStart 处开始，显示在页边以内的 xLeft 和 xRight 之间。你的工作就是计算 xStart ，它是所显示文本在水平方向上的开始处。如果文本用等间距(fixed-pitch)字体来显示，这一工作就能够大大简化，但这只是少数情况。你首先需要计算字符串的文本范围(text extent)：

```
GetTextExtentPoint32 (hdc, szText, strlen (szText), &size) ;
```

如果 size.cx 比(xRight - xLeft)大，那么，这一行对于页边内的空间而言就太长了。让我们假设它没有那么长。

要使文本对齐到左页边，你只需将 xStart 设置成 xLeft 然后输出文本即可：

```
TextOut (hdc, xStart, yStart, szText, strlen (szText)) ;
```

这很简单。接着，你把 size.cy 加到 yStart 上，就可以输出下一行了。

要使文本对齐到右页边，你把 xStart 设置成下面的值：

```
xStart = xRight - size.cx ;
```

要让文本居中，你把 xStart 设置成下面的值：

```
xStart = (xLeft + xRight - size.cx) / 2 ;
```

下面该是困难的工作了——使文本两端对齐。两个页边之间的距离是(xRight - xLeft)，如果不进行两端对齐，那么文本的宽度为 size.cx ，这两个值之间的差是：

```
xRight - xLeft - size.cx
```

这个值必须平均分配给前面那个字符串里的三个空格。这里所需要的处理听起来似乎很不好办，其实不至于。你通过下面的调用来完成它：

```
SetTextJustification (hdc, xRight - xLeft - size.cx, 3) ;
```

第二个参数是将要分配给字符串中空格的空白空间宽度。第三个参数是空格的个数，在这里，它为 3 。

接着，把 xStart 设置成 xLeft 并用 TextOut 输出文本：

```
TextOut (hdc, xStart, yStart, szText, strlen (szText)) ;
```

文本将在 xLeft 和 xRight 页边之间两端对齐。

当你调用 SetTextJustification 时，如果空白不能均匀地分配给空格字符，它就会累积一个误差项，这个误差项将影响后续的 GetTextExtent 调用。所以，在每次开始一个新行时，应该清除这个误差项：

```
SetTextJustification (hdc, 0, 0) ;
```

4.13.9 处理段落

如果你在处理整个段落，你就必须从段首开始扫描空格。每遇到一个空格，你就调用 GetTextExtentPoint32 来确定左右页边之间能否容纳到此为止的文本。如果文本超出了范围，就必须回到前一个空格。这样，就确定了一行中应该显示的字符串。如果你想让它两端对齐，就调用 SetTextJustification 和 TextOut，然后清除误差项，并前进到下一行。

图 4-34 中的 JUSTIFY1 程序用这种方法来显示 Herman Melville 的 Moby Dick 的第一段，它被硬编码为用 15 点的 Times New Roman 字体来显示。不过，你可以改一下程序开头的 MyCreateFont 定义并重新编译一下，就可以换一种字体。你还可以使用程序开始处定义的那些标识符来改变文本对齐方式。

JUSTIFY1.MAK

```
#------------------------
# JUSTIFY1.MAK make file
#------------------------

justify1.exe : justify1.obj ezfont.obj
     $(LINKER) $(GUIFLAGS) -OUT:justify1.exe justify1.obj ezfont.obj $(GUILIBS)

justify1.obj : justify1.c
     $(CC) $(CFLAGS) justify1.c

ezfont.obj : ezfont.c
     $(CC) $(CFLAGS) ezfont.c
```

JUSTIFY1.C

```
/*-----------------------------------------
   JUSTIFY1.C - Justified Type Program
               (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include "ezfont.h"

#define LEFT        0
#define RIGHT       1
#define CENTER      2
#define JUSTIFIED   3

#define ALIGN       JUSTIFIED
```

```
#define MyCreateFont EzCreateFont (hdc, "Times New Roman", 150, 0, 0, TRUE)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static     char szAppName[] = "Justify1" ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW * CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Justified Type",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DrawRuler (HDC hdc, RECT *prc)
     {
     static int iRuleSize [16] = { 360, 72, 144, 72, 216, 72, 144, 72,
                                   288, 72, 144, 72, 216, 72, 144, 72 } ;
     int        i, j ;
     POINT      ptClient ;

     SaveDC (hdc) ;
                    // Set Logical Twips mapping mode

     SetMapMode (hdc, MM_ANISOTROPIC) ;
     SetWindowExtEx (hdc, 1440, 1440, NULL) ;
```

```
     SetViewportExtEx (hdc, GetDeviceCaps (hdc, LOGPIXELSX),
                            GetDeviceCaps (hdc, LOGPIXELSY), NULL) ;

                    // Move the origin to a half inch from upper left

     SetWindowOrgEx (hdc, -720, -720, NULL) ;

                    // Find the right margin (quarter inch from right)

     ptClient.x = prc->right ;
     ptClient.y = prc->bottom ;
     DPtoLP (hdc, &ptClient, 1) ;
     ptClient.x -= 360 ;

                    // Draw the rulers

     MoveToEx (hdc, 0,          -360, NULL) ;
     LineTo   (hdc, ptClient.x, -360) ;
     MoveToEx (hdc, -360,          0, NULL) ;
     LineTo   (hdc, -360, ptClient.y) ;

     for (i = 0, j = 0 ; i <= ptClient.x ; i += 1440 / 16, j++)
          {
          MoveToEx (hdc, i, -360, NULL) ;
          LineTo   (hdc, i, -360 - iRuleSize [j % 16]) ;
          }
     for (i = 0, j = 0 ; i <= ptClient.y ; i += 1440 / 16, j++)
          {
          MoveToEx (hdc, -360, i, NULL) ;
          LineTo   (hdc, -360 - iRuleSize [j % 16], i) ;
          }

     RestoreDC (hdc, -1) ;
     }

void Justify (HDC hdc, PSTR pText, RECT *prc, int iAlign)
     {
     int  xStart, yStart, iBreakCount ;
     PSTR pBegin, pEnd ;
     SIZE size ;

     yStart = prc->top ;
     do                             // for each text line
          {
          iBreakCount = 0 ;
          while (*pText == ' ')
                                    // skip over leading blanks
               pText++ ;
          pBegin = pText ;

          do                        // until the line is known
               {
               pEnd = pText ;

               while (*pText != '\0' && *pText++ != ' ') ;
               if (*pText == '\0')
```

```
                    break ;
                                        // for each space, calculate extents
               iBreakCount++ ;
               SetTextJustification (hdc, 0, 0) ;
               GetTextExtentPoint32 (hdc, pBegin, pText - pBegin - 1, &size) ;
               }
          while ((int) size.cx < (prc->right - prc->left)) ;

          iBreakCount- ;
          while (*(pEnd - 1) == ' ')   // eliminate trailing blanks
               {
               pEnd- ;
               iBreakCount-- ;
               }

          if (*pText == '\0' || iBreakCount <= 0)
               pEnd = pText ;

          SetTextJustification (hdc, 0, 0) ;
          GetTextExtentPoint32 (hdc, pBegin, pEnd - pBegin, &size) ;

          switch (iAlign)              // use alignment for xStart
               {
               case LEFT :
                    xStart = prc->left ;
                    break ;

               case RIGHT :
                    xStart = prc->right - size.cx ;
                    break ;

               case CENTER :
                    xStart = (prc->right + prc->left - size.cx) / 2 ;
                    break ;

               case JUSTIFIED :
                    if (*pText != '\0' && iBreakCount > 0)
                         SetTextJustification (hdc,
                              prc->right - prc->left - size.cx,
                              iBreakCount) ;
                    xStart = prc->left ;
                    break ;
               }

          TextOut (hdc, xStart, yStart, pBegin, pEnd - pBegin) ;
          yStart += size.cy ;
          pText = pEnd ;
          }
     while (*pText && yStart < prc->bottom) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                  LPARAM lParam)
     {
     static char szText[] = "Call me Ishmael. Some years ago - never mind "
                            "How long precisely - having little or no money "
```

```
                        "in my purse, and nothing particular to interest "
                        "me on shore, I thought I would sail about a "
                        "little and see the watery part of the world. It "
                        "is a way I have of driving off the spleen, and "
                        "regulating the circulation.  Whenever I find "
                        "myself growing grim about the mouth; whenever "
                        "it is a damp, drizzly November in my soul; "
                        "whenever I find myself involuntarily pausing "
                        "before coffin warehouses, and bringing up the "
                        "rear of every funeral I meet; and especially "
                        "whenever my hypos get such an upper hand of me, "
                        "that it requires a strong moral principle to "
                        "prevent me from deliberately stepping into the "
                        "street, and methodically knocking people   hats "
                        "off - then, I account it high time to get to sea "
                        "as soon as I can. This is my substitute for "
                        "pistol and ball.  With a philosophical flourish "
                        "Cato throws himself upon his sword; I quietly "
                        "take to the ship. There is nothing surprising "
                        "in this. If they but knew it, almost all men in "
                        "their degree, some time or other, cherish very "
                        "nearly the same feelings towards the ocean with "
                        "me." ;

     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rcClient ;

     switch (iMsg)
          {
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rcClient) ;
               DrawRuler (hdc, &rcClient) ;

               rcClient.left  += GetDeviceCaps (hdc, LOGPIXELSX) / 2 ;
               rcClient.top   += GetDeviceCaps (hdc, LOGPIXELSY) / 2 ;
               rcClient.right -= GetDeviceCaps (hdc, LOGPIXELSX) / 4 ;

               SelectObject (hdc, MyCreateFont) ;

               Justify (hdc, szText, &rcClient, ALIGN) ;

               DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT)));
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 4-34 JUSTIFY1程序

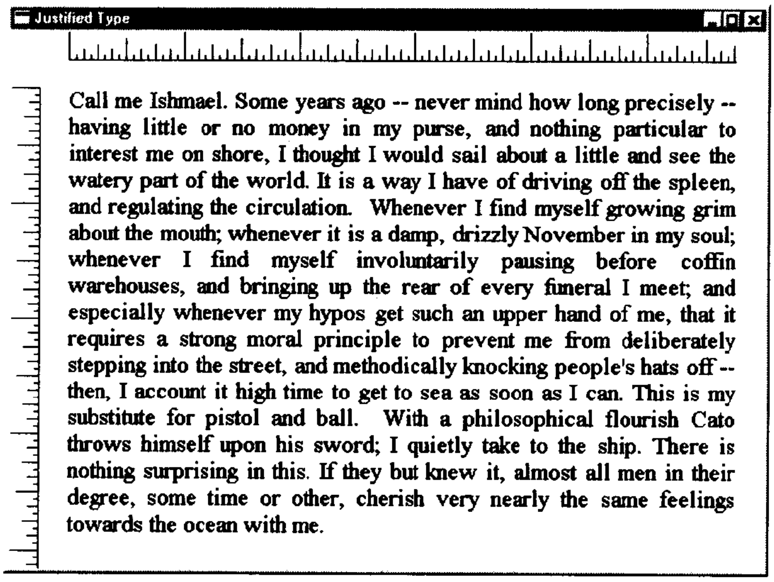


图 4-35 JUSTIFY1显示

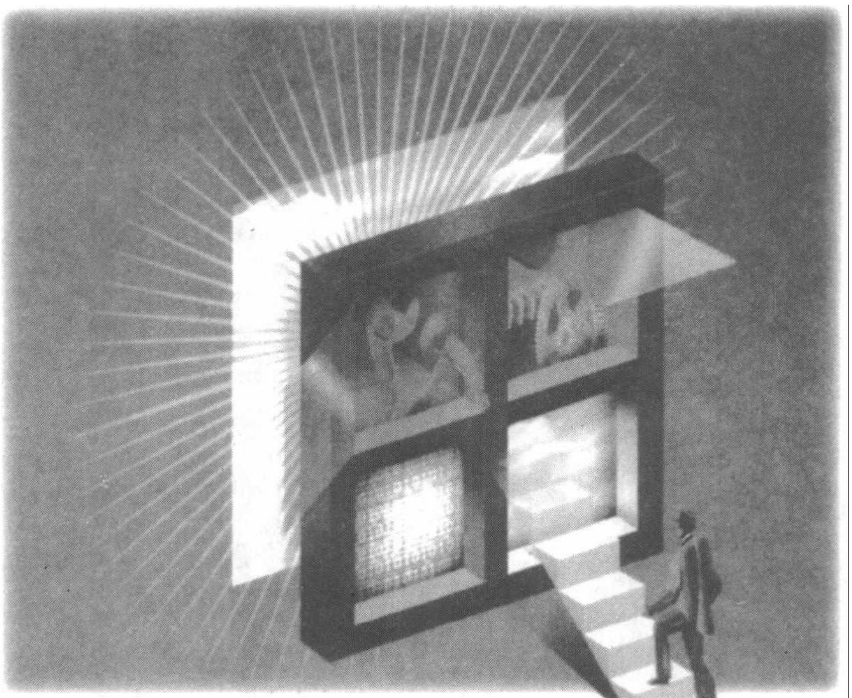
JUSTIFY1 在客户区的上方和左方各显示一个标尺，当然，是以逻辑英寸为单位，DrawRuler 函数绘制标尺。一个矩形结构被用来指定文本在其中两端对齐的区域。

主要的格式化文本的工作由 Justify 函数完成。JUSTIFY1 从文本的开头向后搜索空格，并使用 GetTextExtentPoint32 来度量每一行。如果行的长度超过了显示区域的宽度，JUSTIFY1 就回到前一个空格，并采用到该空格为止的那一行。依赖于 ALIGN 常量的值，这一行被左对齐、右对齐、居中和两端对齐。

JUSTIFY1 并不是完美的，具体说，如果一行中的单词不到两个，它的两端对齐处理过程就不灵了。就算我们解决了这个问题（这并不太难），当一个单词太长以至于一行都放不下时，程序还是会出错。当然，如果你要编写在同一行上使用多种字体的程序（这就是 Windows 字处理器所做的看起来容易的事），事情就更复杂了。还好，没人会说这样的事情容易，它只是要比你自己亲自来做这些处理容易一些。

第二部分

输入手段



第5章 键 盘

像大多数在个人计算机上运行的交互式程序一样，键盘也是 Windows 应用程序的一个重要的输入手段。虽然 Windows 支持鼠标输入设备，但是键盘仍然占据主宰地位。当然，因为一些比较老的 PC 用户更喜欢使用键盘，所以建议开发人员设法实现键盘对程序功能的全面支持（当然，在有些情况下，如绘图程序或者桌面排版程序，只用键盘是不现实的，还要求有鼠标）。

键盘不能被视为独立于应用程序其它功能的输入设备。例如，程序经常在窗口的客户区显示输入的字符来响应键盘输入。因此，键盘的输入与文本的显示必须一起处理。如果让你的程序适合于外国语言和市场对你来说非常重要，那么你还需要知道一些关于 Windows 95 对扩展 ASCII 字符集（代码大于或者等于 128）和双字节字符集（DBCS）的支持，以及 Windows NT 对 Unicode 的 16 位键盘编码的支持。

5.1 键盘基础

你可以猜到，Windows 基于消息的体系结构很适于键盘的使用，你的程序通过把消息传递给你的窗口过程来获知击键。

事实上比这要复杂一些：当用户按下或者释放一个键时，键盘驱动程序将击键传给 Windows，Windows 在系统消息队列中保存这一击键（以消息的形式），然后一次一个消息地将击键传送到拥有“输入焦点”（我马上将加以说明）的程序的消息队列，程序再将这些消息分发给适当的窗口过程。

采用这种两步处理——首先将消息保存在系统消息队列中然后再传递给应用程序消息队列——是由于同步问题。当用户的击键速度比程序处理这些击键的速度还要快时，Windows 会在系统消息队列中保存一些特殊击键，这些击键可能会导致输入焦点被切换到另一个程序中，而使得接下来的击键应该被传递给另一个程序。Windows 正确地同步了这些键盘消息。

Windows 给程序发送八种不同的消息来代表不同的键盘事件，这似乎显得太多了，但是你的程序可以忽略其中的许多消息而不会有任何问题。并且，在大多数情况下，这些消息中包含的键盘信息会多于你的程序所需要的。处理键盘的一部分工作就是识别出哪些信息是重要的，哪些是不重要的。

5.1.1 忽略键盘

虽然键盘是 Windows 程序最基本的输入资源，但是程序不必对它接收的所有消息都作

出响应。 Windows 本身也能处理许多键盘功能，例如，你可以忽略那些属于系统功能的击键，它们通常用到 Alt 键。

程序不必监视这些击键，因为 Windows 会将击键的作用通知程序（当然，如果程序想这么做，它也能监视这些击键）。例如，如果 Windows 用户使用键盘选择一个菜单项，Windows 就会给程序发送一条“菜单项已经被选中”的消息，而不管它是由鼠标还是由键盘选择的（在第 10 中将介绍菜单）。

有些 Windows 程序使用“键盘加速键”来激发通用菜单项。加速键通常与功能键或者字母同 Ctrl 键的组合键有关，这些键盘加速键在程序的资源描述文件中定义（第 10 章说明如何将这些加速键转换为菜单命令消息，你不必自己去进行转换）。

对话框（第 11 章介绍）也有键盘接口，但是当对话框活动时，应用程序通常不必监视键盘。键盘接口由 Windows 处理， Windows 把关于击键效果的消息发送给程序。对话框可以包含“编辑”控制，以用于文本输入。对话框一般是一个小方框，用户可以在框中输入字符串。 Windows 处理所有编辑控制逻辑，并在输入完毕后，将编辑控制的最后内容传送给程序。

即使在主窗口中，你也可以定义用作编辑控制的子窗口。一个极端的例子是第 8 章中的 POPPAD 程序，这个程序有点像一个大型的多行编辑控制，它依赖于 Windows 来处理所有的零碎工作。

5.1.2 焦点，焦点，谁获得了焦点

键盘必须由所有在 Windows 下运行的应用程序共享。有些应用程序可能有多个窗口，键盘必须由这些属于同一应用程序的窗口共享。当在键盘上按下一个键时，只有一个窗口过程能接收到“该键已被按下”的消息，接收到这个键盘消息的窗口被称为具有“输入焦点”的窗口。

输入焦点的概念与“活动窗口”的概念紧密相关。有输入焦点的窗口可能是活动窗口，也可能是活动窗口的子窗口。活动窗口很容易辨别。如果活动窗口有标题条，那么 Windows 就加亮它的标题条。如果活动窗口没有标题条而有对话框边框（对话框中很常见的一种形式），那么 Windows 就加亮该边框。如果活动窗口被最小化了，那么 Windows 就加亮标题条的文字。

最常见的子窗口是按钮、单选按钮、复选框、滚动条、编辑和列表框等控制。子窗口本身并不是活动窗口。如果活动子窗口拥有输入焦点，那么其父窗口就是活动窗口。子窗口控制通常使用闪烁的光标来指示它们拥有输入焦点。

如果活动窗口最小化了，那么没有一个窗口拥有输入焦点。 Windows 仍然发送键盘消息给图标，但是这些消息与发送给非最小化窗口的键盘消息有不同的形式。

窗口过程通过捕获 WM_SETFOCUS 和 WM_KILLFOCUS 消息来判定它何时拥有输入焦点。 WM_SETFOCUS 指示窗口正在得到输入焦点，而 WM_KILLFOCUS 表示窗口正在失去输入焦点。

5.1.3 击键和字符

应用程序从 Windows 接收的关于键盘事件的消息可以分为击键和字符两类，这同用户看待键盘的两种方式相一致。首先，你可以将键盘看作是键的集合。键盘只有唯一的 A 键，按下该键是一次击键，释放该键也是一次击键。但是键盘也是能产生可显示字符的输入设备。根据 Ctrl 、 Shift 和 Caps Lock 键的状态， A 键能产生几个字符。通常情况下，此字符为小写 a 。如果按下 Shift 键或者切换到 Caps Lock ，就变成大写 A 。如果 Ctrl 按下了，则该字符为 Ctrl-A 。在外国语键盘上， A 击键之前可能有“死字符键（ dead-character key ）”或者 Shift 、 Ctrl 或者 Alt 的不同组合，这些组合可以产生带有音调标志的小写 a 或者大写 A 。

对产生可显示字符的击键组合， Windows 不仅给程序发送击键消息，而且还发送字符消息。有些键不产生字符，这些键包括 Shift 键、功能键、光标移动键和特殊字符键如 Insert 和 Delete 。对这些键， Windows 只产生击键消息。

5.2 击键消息

当你按下一个键时， Windows 把 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息中的一个放进有输入焦点的窗口队列；当你释放一个键时， Windows 把 WM_KEYUP 或者 WM_SYSKEYUP 消息放进队列中。

	键按下	键释放
非系统键	WM_KEYDOWN	WM_KEYUP
系统键	WM_SYSKEYDOWN	WM_SYSKEYUP

通常“ down （按下）”和“ up （放开）”消息是成对出现的。不过，如果你按住一个键以致自动重复功能生效，那么 Windows 会给窗口过程发送一系列 WM_KEYDOWN（或者 WM_SYSKEYDOWN ）消息和一个的 WM_KEYUP （或者 WM_SYSKEYUP ）消息（该键最后被释放时）。像所有入队的消息一样，击键消息也有时间差。通过调用 GetMessageTime ，你可以获得按下或者释放一键的相对时间。

5.2.1 系统击键与非系统击键

WM_SYSKEYDOWN 和 WM_SYSKEYUP 中的“ SYS ”代表“系统”，它表示该击键对 Windows 比对 Windows 应用程序更加重要。 WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息经常由与 Alt 相组合的击键产生，这些击键激活程序菜单或者系统菜单上的选项，或者用于切换活动窗口等系统功能（ Alt-Tab 或者 Alt-Esc ），也可以用作系统菜单加速键（ Alt 键与一个功能键相结合）。程序通常忽略 WM_SYSKEYUP 和

WM_SYSKEYDOWN 选项，并将它们传送到 DefWindowProc 。由于 Windows 要处理所有 Alt 键的功能，所以你无需俘获这些消息。你的窗口过程将最后收到关于这些击键结果（如菜单选择）的其它消息。如果你想在自己的窗口过程中包括捕获系统击键的代码（如本章后面的KEYLOOK程序所作的一样），那么在处理这些消息之后再传送到DefWindowProc，Windows 就仍然可以将它们用于通常的目的。

但是，请再考虑一下，几乎所有会影响用户程序窗口的消息都会先通过用户窗口过程。只有用户把消息传送到 DefWindowProc ， Windows 才会对消息进行处理。例如，如果你加入下面几行语句：

```
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
    return 0;
```

到一个窗口过程中，那么当你的程序拥有输入焦点时，就可以有效地禁止对所有 Alt 键操作（菜单命令、 Alt_Tab 、 Alt_Esc 等）的使用。虽然我怀疑你是否会这么做，但是，我相信你会感到窗口过程的强大功能。

WM_KEYDOWN 和 WM_KEYUP 消息通常是在按下或者释放一个不带 Alt 键的键时产生的，你的程序可以使用或者忽略这些消息， Windows 本身并不处理这些消息。

5. 2. 2 lParam 变量

对所有四类击键消息，传送到窗口过程的 32 位的 lParam 变量划分为 6 个域：重复记数、 OEM 扫描码、扩展键标志、环境代码、键的前一状态以及转换状态（见图 5-1 ）。

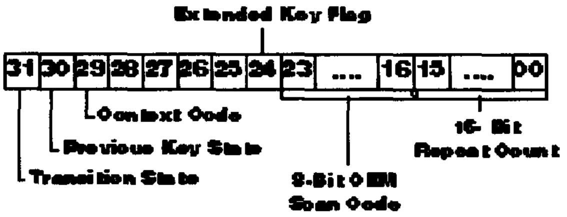


图 5-1 lParam变量的6个击键消息域

5.2.2.1 重复记数

重复记数是该消息所表示的击键次数，大多数情况下，重复记数设置为 1 。不过，如果按下一个键之后，你的窗口过程不够快，以致不能处理自动重复速率（缺省时近似为每秒 10 个字符）下的按键消息， Windows 就把几个 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息组合到单个消息中并相应地增加重复记数。 WM_KEYUP 和 WM_SYSKEYUP 消息的重复记数总为 1 。

因为重复记数大于 1 指示击键速率大于你程序的处理能力，所以你也可能想在处理键盘消息时忽略重复记数。几乎每个人都有滚屏过快的经历，因为多余的击键堆满了键盘缓

冲区，所以忽略你程序中的重复记数将明显地减少滚屏过快的可能性。不过，有时可能也会用到重复记数。你应该尝试使用两种方法运行程序，从中找出一种较好的方法。

5.2.2.2 OEM扫描码

OEM 扫描码是由计算机硬件产生的键盘扫描码（如果你熟悉汇编语言编程，这个扫描码就是 BIOS 16H 中断调用从寄存器 AH 中返回给程序的那个值）。Windows 应用程序通常忽略 OEM 扫描码，因为有一种更好地编码键盘消息的方法。

5.2.2.3 扩展键标志

如果击键结果来自 IBM 增强键盘的附加键之一，那么扩展键标志为 1（IBM 增强型键盘的顶端有功能键，另外还有独立的[结合的]光标-数字小键盘）。对键盘右端的 Alt 和 Ctrl 键，以及不是数字小键盘那部分的光标移动键（包括 Insert 和 Delete 键）、数字小键盘上的斜杠（/）和 Enter 键以及 Num Lock 键等，此标志均被设置为 1。Windows 程序通常忽略扩展标志。

5.2.2.4 环境代码

环境代码在按下 Alt 键后为 1。对 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息，这一位总是 1；对 WM_KEYUP 和 WM_KEYDOWN 消息，这一位总是 0；但是有两个例外：

- 如果活动窗口最小化了，则它没有输入焦点。这时候所有的击键都产生 WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息。如果 Alt 键未被按下，则环境代码域被设置为 0（Windows 使用 SYS 键盘消息，从而使最小化了的活动窗口不处理这些击键）。
- 对于一些外国语键盘，有些字符是通过 Shift、Ctrl 或者 Alt 键与其它键相组合产生的，因此伴随 WM_KEYUP 和 WM_KEYDOWN 消息的 lParam 变量的环境代码域为 1，但是此消息并非系统击键消息。

5.2.2.5 键的先前状态

如果在此之前键是释放的，则键的先前状态为 0，否则为 1。对 WM_KEYUP 或者 WM_SYSKEYUP 消息，它总是设置为 1；但是对 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息，此位可以为 0，也可以为 1。如果为 1，则表示该键是自动重复功能所产生的第二条或者后续消息。

5.2.2.6 转换状态

如果键正被按下，则转换状态为 0；如果键正被释放，则转换状态为 1。对 WM_KEYDOWN 或者 WM_SYSKEYDOWN 消息，此域为 0；对 WM_KEYUP 或者 WM_SYSKEYUP 消息，此域为 1。

5.2.3 虚拟键码

虽然 lParam 中的一些信息对处理 WM_KEYUP、WM_KEYDOWN、WM_SYSKEYUP 和 WM_SYSKEYDOWN 消息都是有用的，但是 wParam 参数更加重要，这个参数含有标识哪个键被按下或者被释放的“虚拟键码”。Windows 的开发人员使用独立于设备的方法定义虚拟键。由于这个原因，一些虚拟键码不能在 IBM PC 和与之严格兼容的机器上产生，但是在其它制造商的键盘上是可行的。

虚拟键码

十进制	十六进制	WINDOWS.H 标识符	必需?	IBM 键盘
1	01	VK_LBUTTON		
2	02	VK_RBUTTON		
3	03	VK_CANCEL	√	Ctrl-Break
4	04	VK_MBUTTON		
8	08	VK_BACK	√	Backspace
9	09	VK_TAB	√	Tab
12	0C	VK_CLEAR		Num Lock 关闭时的数字键盘 5
13	0D	VK_RETURN	√	Enter
16	10	VK_SHIFT	√	Shift
17	11	VK_CONTROL	√	Ctrl
18	12	VK_MENU	√	Alt
19	13	VK_PAUSE		Pause
20	14	VK_CAPITAL	√	Caps Lock
27	1B	VK_ESCAPE	√	Esc
32	20	VK_SPACE	√	Spacebar
33	21	VK_PRIOR	√	Page Up
34	22	VK_NEXT	√	Page Down
35	23	VK_END		End
36	24	VK_HOME	√	Home
37	25	VK_LEFT	√	Left Arrow
38	26	VK_UP	√	Up Arrow
39	27	VK_RIGHT	√	Right Arrow
40	28	VK_DOWN	√	Down Arrow
41	29	VK_SELECT		
42	2A	VK_PRINT		
43	2B	VK_EXECUTE		
44	2C	VK_SNAPSHOT		Print Screen
45	2D	VK_INSERT	√	Insert
46	2E	VK_DELETE	√	Delete

续表

十进制	十六进制	WINDOWS.H 标识符	必需?	IBM 键盘
47	2F	VK_HELP		
48-57	30-39		√	主键盘上的 0 到 9
65-90	41-5A		√	A through Z
96	60	VK_NUMPAD0		Num Lock 打开时的数字键盘 0
97	61	VK_NUMPAD1		Num Lock 打开时的数字键盘 1
98	62	VK_NUMPAD2		Num Lock 打开时的数字键盘 2
99	63	VK_NUMPAD3		Num Lock 打开时的数字键盘 3
100	64	VK_NUMPAD4		Num Lock 打开时的数字键盘 4
101	65	VK_NUMPAD5		Num Lock 打开时的数字键盘 5
102	66	VK_NUMPAD6		Num Lock 打开时的数字键盘 6
103	67	VK_NUMPAD7		Num Lock 打开时的数字键盘 7
104	68	VK_NUMPAD8		Num Lock 打开时的数字键盘 8
105	69	VK_NUMPAD9		Num Lock 打开时的数字键盘 9
106	6A	VK_MULTIPLY		数字键盘*
107	6B	VK_ADD		数字键盘+
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		数字键盘-
110	6E	VK_DECIMAL		数字键盘.
111	6F	VK_DIVIDE		数字键盘/
112	70	VK_F1	√	功能键 F1
113	71	VK_F2	√	功能键 F2
114	72	VK_F3	√	功能键 F3
115	73	VK_F4	√	功能键 F4
116	74	VK_F5	√	功能键 F5
117	75	VK_F6	√	功能键 F6
118	76	VK_F7	√	功能键 F7
119	77	VK_F8	√	功能键 F8
120	78	VK_F9	√	功能键 F9
121	79	VK_F10	√	功能键 F10
122	7A	VK_F11		功能键 F11(增强型键盘)
123	7B	VK_F12		功能键 F12(增强型键盘)
124	7C	VK_F13		
125	7D	VK_F14		
126	7E	VK_F15		
127	7F	VK_F16		
144	90	VK_NUMLOCK		Num Lock
145	91	VK_SCROLL		Scroll Lock

在"必需?"一列中的复选标记(√)表示该键对 Windows 的所有实现都是必需的。Windows 还要求键盘和键盘驱动程序允许 Shift、Ctrl、Shift+Ctrl 键同所有字母键、所有必需的光标键、所有必需的功能键组合使用。VK_LBUTTON、VK_MBUTTON、VK_RBUTTON 虚拟键码指鼠标的左、中、右按钮。不过,你不会收到其 wParam 设置为这些值的击键消息,鼠标会产生它自己的消息,我们将在下一章中看到。

5.2.4 换档状态

伴随 WM_KEYDOWN、WM_KEYUP、WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息的 wParam 和 lParam 参数并没有将换档键的状态告诉程序。用 GetKeyState 函数可以得到任意虚拟键的当前状态,此函数经常用于获取换档键(Shift、Ctrl 和 Alt)和开关键(Caps Lock、Num Lock 和 Scroll Lock)的状态。例如:

```
GetKeyState (VK_SHIFT) ;
```

在按下 Shift 键时将返回一个负值(即高位被置位)。如果 Caps Lock 键打开,则从

```
GetKeyState (VK_CAPITAL) ;
```

返回的值的低位被置位。使用虚拟键码 VK_LBUTTON、VK_RBUTTOM 和 VK_MBUTTON 也可以得到鼠标按钮的状态。不过,大多数需要监视鼠标按钮与击键相组合的 Windows 应用程序都使用其它的方法来做到这一点——即在接收到鼠标消息时检查击键。实际上,换档状态信息包含在鼠标信息中(正如你在下一章中将看到的一样)。

请注意 GetKeyState 的使用,它并非实时检查键盘状态,而只是检查直到并且包括当前正在处理的消息的键盘状态。GetKeyState 不允许你检索与正常键盘消息无关的键盘信息。例如,你或许想暂停窗口过程中的处理,直到你按下 F1 功能键为止:

```
while (GetKeyState (VK_F1) >= 0) ; // 错误!!!!
```

不要这么做!你的程序必须在 GetKeyState 检测键的状态之前从队列中检索键盘消息。这种同步处理是为你着想,因为如果你希望知道特定击键消息的换档状态,那么 GetKeyState 肯定是准确的,即使在换档键被释放以后才处理消息也是这样。如果你的确需要使用键的当前状态,那么可以用 GetAsyncKeyState.

5.2.5 使用击键消息

如果程序能够获得关于每个击键的信息,这当然很理想,但是大多数 Windows 程序忽略了几乎所有的击键,而只处理几条击键消息。WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息是由 Windows 系统函数使用的,你不必为此费心,就算你要处理 WM_KEYDOWN 消息,通常也可以忽略 WM_KEYUP 消息。

Windows 程序通常为不产生字符的击键使用 WM_KEYDOWN 消息。虽然你可以认为

借助击键消息和换档键状态信息能将击键消息转换为字符消息，但是不要这么做，因为你将遇到国际键盘间的不同所带来的问题。例如，如果你得到 wParam 等于 33H 的 WM_KEYDOWN 消息，你就可以知道用户按下了键 3 ，到此为止一切正常。这时，如果用 GetKeyState 发现 Shift 键被按下，你可能会认为用户输入了 # 号，这可不一定。比如，一个英国用户就是在输入一个 £ 。因此，对于光标移动键、功能键、 Insert 和 Delete 等特殊键， WM_KEYDOWN 消息是最有用的。不过， Insert 、 Delete 和功能键经常作为菜单加速键。因为 Windows 能把菜单加速键翻译为菜单命令消息，所以你就不必自己来处理击键。 PC 机上的一些非 Windows 程序大量使用功能键与 Shift 、 Ctrl 和 Alt 键的组合，在 Windows 应用程序中也可以如此，但并不推荐这样作。如果你确实希望使用功能键，那么这些键应该是重复菜单命令。 Windows 的目标之一是提供不需要记忆或者使用复杂命令流程的用户界面。

我们已经尽可能消除对击键消息的处理，只有一种情形例外：那就是在大多数情况下，你要为光标移动键处理 WM_KEYDOWN 消息。当使用光标键时，你通过 GetKeyState 来检查 Shift 键和 Ctrl 键的状态。 Windows 函数经常使用 Shift 与光标键的组合键来扩大字处理文档里选中的范围， Ctrl 键常用于修改光标键的意义（例如， Ctrl 与右箭头键相组合可以表明光标右移一个字）。

决定如何使用键盘的最佳方法之一是了解现有的 Windows 程序是如何使用键盘的。如果你不喜欢那些定义，当然可以对其加以修改，但是这样做不利于其它人很快学会使用你的程序。

5.3 增强 SYSMETS ：增加键盘接口

当编写第三章中三个版本的 SYSMETS 程序时，我们还不了解键盘，只能使用滚动条和鼠标来滚动文本。现在我们知道了如何处理键盘，那么不妨在 SYSMETS 中加入键盘接口。显然，键盘接口的工作之一是处理光标移动键。我们将大多数光标键（如 Home 、 End 、 Page Up 、 Page Down 、 Up Arrow 和 Down Arrow ）用于垂直滚动， Left Arrow 和 Right Arrow 键用于水平滚动。

5.3.1 加入 WM_KEYDOWN 逻辑

创建键盘接口的一种简单方法是在窗口过程中加入与 WM_VSCROLL 和 WM_HSCROLL 逻辑相平行的 WM_KEYDOWN 逻辑:

```
case WM_KEYDOWN :
     iVscrollInc = iHscrollInc = 0 ;

     switch (wParam)
          {
          case VK_HOME :          // same as WM_VSCROLL, SB_TOP
               iVscrollInc = -iVscrollPos ;
               break ;
```

```
          case VK_END :          // same as WM_VSCROLL, SB_BOTTOM
               iVscrollInc = iVscrollMax - iVscrollPos ;
               break ;

          case VK_UP :           // same as WM_VSCROLL, SB_LINEUP
               iVscrollInc = -1 ;
               break ;

          case VK_DOWN :         // same as WM_VSCROLL, SB_LINEDOWN
               iVscrollInc = 1 ;
               break ;

          case VK_PRIOR :        // same as WM_VSCROLL, SB_PAGEUP
               iVscrollInc = min (-1, -cyClient / cyChar) ;
               break ;

          case VK_NEXT :         // same as WM_VSCROLL, SB_PAGEDOWN
               iVscrollInc = max (1, cyClient / cyChar) ;
               break ;

          case VK_LEFT :         // same as WM_HSCROLL, SB_PAGEUP
               iHscrollInc = -8 ;
               break ;

          case VK_RIGHT :        // same as WM_HSCROLL, SB_PAGEDOWN
               iHscrollInc = 8 ;
               break ;

          default :
               break ;
          }

     if (iVscrollInc = max (-iVscrollPos,
               min (iVscrollInc, iVscrollMax - iVscrollPos)))
          {
          iVscrollPos += iVscrollInc ;
          ScrollWindow (hwnd, 0, -cyChar * iVscrollInc, NULL, NULL) ;
          SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
          UpdateWindow (hwnd) ;
          }

     if (iHscrollInc = max (-iHscrollPos,
               min (iHscrollInc, iHscrollMax - iHscrollPos)))
          {
          iHscrollPos += iHscrollInc ;
          ScrollWindow (hwnd, -cxChar * iHscrollInc, 0, NULL, NULL) ;
          SetScrollPos (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
          }

     return 0 ;
```

你是否也和我一样讨厌这个代码？简单地复制滚动条代码是不明智的，因为如果我们要修改滚动条代码，那么必须相应地修改 WM_KEYDOWN 逻辑。应该为此寻找一种更好的方法，这种方法是存在的。

5.3.2 发送消息

只是将每一种WM_KEYDOWN消息翻译成等价的WM_VSCROLL和WM_HSCROLL消息，然后把这些消息传给WndProc，这种方法是不是好些呢？对WndProc来说，它会以为自己得到了滚动条消息WM_VSCROLL或WM_SCROLL。在Windows中，这种方法是可行的。发消息的这个函数叫做SendMessage，它所用的参数与传递到窗口过程的参数是相同的：

```
SendMessage (hwnd, message, wParam, lParam) ;
```

当调用SendMessage时，Windows调用窗口句柄为hwnd的窗口过程，并且把这四个参数传给它。当窗口过程完成消息处理之后，Windows把控制返回到SendMessage调用之后的下一条语句。你发送消息到其中的窗口过程可以是同一窗口过程、同一程序中的另一个窗口过程或者另一个应用程序中的窗口过程。

下面说明在SYSMETS程序中如何使用SendMessage处理WM_KEYDOWN消息：

```
case WM_KEYDOWN :
     switch (wParam)
          {
          case VK_HOME :
               SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0L) ;
               break ;

          case VK_END :
               SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0L) ;
               break ;

          case VK_PRIOR :
               SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0L) ;
               break ;

          [其它程序行]
```

至此你已经有了总的印象。我们的目标是为滚动条增加键盘接口，并且也正是在这么做的。通过把滚动条消息发送到窗口过程，我们实现了用光标移动键重复滚动条的功能。现在你知道在SYSMETS3中为WM_VSCROLL消息引入对SB_TOP和SB_BOTTOM的处理的原因了。在那里并没有用到它，但是现在处理Home和End键时就有用了。如图5-2所示的SYSMETS的最后版本中就体现了这些变化。编译这个程序时还需要第三章中的SYSMETS.H文件（图3-4）。

SYSMETS.MAK

```
#-----------------------
# SYSMETS.MAK make file
#-----------------------
```

```
sysmets.exe : sysmets.obj
     $(LINKER) $(GUIFLAGS) -OUT:sysmets.exe sysmets.obj $(GUILIBS)

sysmets.obj : sysmets.c sysmets.h
     $(CC) $(CFLAGS) sysmets.c
```

SYSMETS.C

```
/*-----------------------------------------------------
   SYSMETS.C -- System Metrics Display Program (Final)
                (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "sysmets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "System Metrics",
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
```

```
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth,
                 iVscrollPos, iVscrollMax, iHscrollPos, iHscrollMax ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i, x, y, iPaintBeg, iPaintEnd, iVscrollInc,
                 iHscrollInc ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;

               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;

               ReleaseDC (hwnd, hdc) ;

               iMaxWidth = 40 * cxChar + 22 * cxCaps ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               iVscrollMax = max (0, NUMLINES + 2 - cyClient / cyChar) ;
               iVscrollPos = min (iVscrollPos, iVscrollMax) ;

               SetScrollRange (hwnd, SB_VERT, 0, iVscrollMax, FALSE) ;
               SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;

               iHscrollMax = max (0, 2 + (iMaxWidth - cxClient) / cxChar) ;
               iHscrollPos = min (iHscrollPos, iHscrollMax) ;

               SetScrollRange (hwnd, SB_HORZ, 0, iHscrollMax, FALSE) ;
               SetScrollPos   (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
               return 0 ;

          case WM_VSCROLL :
               switch (LOWORD (wParam))
                    {
                    case SB_TOP :
                         iVscrollInc = -iVscrollPos ;
                         break ;

                    case SB_BOTTOM :
```

```
                    iVscrollInc = iVscrollMax - iVscrollPos ;
                    break ;

               case SB_LINEUP :
                    iVscrollInc = -1 ;
                    break ;

               case SB_LINEDOWN :
                    iVscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    iVscrollInc = min (-1, -cyClient / cyChar) ;
                    break ;

               case SB_PAGEDOWN :
                    iVscrollInc = max (1, cyClient / cyChar) ;
                    break ;

               case SB_THUMBTRACK :
                    iVscrollInc = HIWORD (wParam) - iVscrollPos ;
                    break ;

               default :
                    iVscrollInc = 0 ;
               }
          iVscrollInc = max (-iVscrollPos,
                        min (iVscrollInc, iVscrollMax - iVscrollPos)) ;

          if (iVscrollInc != 0)
               {
               iVscrollPos += iVscrollInc ;
               ScrollWindow (hwnd, 0, -cyChar * iVscrollInc, NULL, NULL) ;
               SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
               UpdateWindow (hwnd) ;
               }
          return 0 ;

     case WM_HSCROLL :
          switch (LOWORD (wParam))
               {
               case SB_LINEUP :
                    iHscrollInc = -1 ;
                    break ;

               case SB_LINEDOWN :
                    iHscrollInc = 1 ;
                    break ;

               case SB_PAGEUP :
                    iHscrollInc = -8 ;
                    break ;

               case SB_PAGEDOWN :
                    iHscrollInc = 8 ;
                    break ;
```

```
               case SB_THUMBPOSITION :
                    iHscrollInc = HIWORD (wParam) - iHscrollPos ;
                    break ;

               default :
                    iHscrollInc = 0 ;
               }
          iHscrollInc = max (-iHscrollPos,
                        min (iHscrollInc, iHscrollMax - iHscrollPos)) ;

          if (iHscrollInc != 0)
               {
               iHscrollPos += iHscrollInc ;
               ScrollWindow (hwnd, -cxChar * iHscrollInc, 0, NULL, NULL) ;
               SetScrollPos (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
               }
          return 0 ;

     case WM_KEYDOWN :
          switch (wParam)
               {
               case VK_HOME :
                    SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0L) ;
                    break ;

               case VK_END :
                    SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0L) ;
                    break ;

               case VK_PRIOR :
                    SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0L) ;
                    break ;

               case VK_NEXT :
                    SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0L) ;
                    break ;

               case VK_UP :
                    SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0L) ;
                    break ;

               case VK_DOWN :
                    SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0L) ;
                    break ;

               case VK_LEFT :
                    SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0L) ;
                    break ;

               case VK_RIGHT :
                    SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0L) ;
                    break ;
               }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;
```

```
            iPaintBeg = max (0, iVscrollPos + ps.rcPaint.top / cyChar - 1) ;
            iPaintEnd = min (NUMLINES,
                             iVscrollPos + ps.rcPaint.bottom / cyChar) ;

            for (i = iPaintBeg ; i < iPaintEnd ; i++)
                 {
                 x = cxChar * (1 - iHscrollPos) ;
                 y = cyChar * (1 - iVscrollPos + i) ;

                 TextOut (hdc, x, y,
                          sysmetrics[i].szLabel,
                          strlen (sysmetrics[i].szLabel)) ;

                 TextOut (hdc, x + 22 * cxCaps, y,
                          sysmetrics[i].szDesc,
                          strlen (sysmetrics[i].szDesc)) ;

                 SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

                 TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y,
                          szBuffer,
                          wsprintf (szBuffer, "%5d",
                               GetSystemMetrics (sysmetrics[i].iIndex))) ;

                 SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                 }

            EndPaint (hwnd, &ps) ;
            return 0 ;

       case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
       }

  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
  }
```

图 5-2 SYSMETS程序

5.4 字符消息

前面讨论了利用换档状态信息把击键消息翻译为字符消息的方法，并且提到，仅利用转换状态信息还不够，因为还需要知道同国家有关的键盘配置。由于这个原因，你不应该试图把击键消息翻译为字符代码。

Windows 会为你完成这一工作，在前面我们曾看到过这一代码：

```
while (GetMessage (&msg, NULL, 0, 0))
    {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
    }
```

这是 WinMain 中典型的消息循环。GetMessage 函数用队列中的下一个消息填充 msg 结构的域，DispatchMessage 以此消息为参数调用适当的窗口过程。

在这两个函数之间是 TranslateMessage 函数，它将击键消息转换为字符消息。如果消息为 WM_KEYDOWN 或者 WM_SYSKEYDOWN,并且击键与换档状态相组合产生一个字符，则 TranslateMessage 把字符消息放入消息队列中。此字符将是 GetMessage 从消息队列中得到的击键消息之后的下一个消息。

字符消息可以分为四类：

	字符	死字符
非系统字符：	WM_CHAR	WM_DEADCHAR
系统字符：	WM_SYSCHAR	WM_SYSDEADCHAR

WM_CHAR 和 WM_DEADCHAR 消息是从 WM_KEYDOWN 得到的；而 WM_SYSCHAR 和 WM_SYSDEADCHAR 消息是从 WM_SYSKEYDOWN 消息得到的。大多数情况下，Windows 程序会忽略除 WM_CHAR 之外的任何消息。传递到窗口过程的字符代码消息与产生字符代码消息的击键消息有相同的 lParam 参数，wParam 参数是字符的 ASCII 代码。

字符消息是夹在击键消息之间传送到窗口过程的。例如，如果 Caps Lock 未被按下，而用户按下再释放 A 键，窗口过程将接收到下面三条消息：

消息	键或者代码
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 码 a
WM_KEYUP	虚拟键 A

如果你按下 SHIFT 键，再按下 A 键，然后释放 A 键，再释放 SHIFT 键，就会输入大写的 A，而窗口过程会接收到下面五条消息：

消息	键或者代码
WM_KEYDOWN	虚拟键 VK_SHIFT
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 代码 A
WM_KEYUP	虚拟键 A
WM_KEYUP	虚拟键 VK_SHIFT

Shift 键本身不产生字符消息。

如果用户按住 A 键，以使自动重复产生一系列的击键，那么对每条 WM_KEYDOWN 消息，都会得到一条字符消息：

消息	键或者代码
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 代码 a
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 代码 a
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 代码 a
WM_KEYDOWN	虚拟键 A
WM_CHAR	ASCII 代码 a
WM_KEYUP	虚拟键 A

如果某些 WM_KEYDOWN 消息的重复记数大于 1，那么相应的 WM_CHAR 消息将具有同样的重复记数。

组合使用 Ctrl 键与字母键会产生 01H （Ctrl-A）到 1AH （Ctrl-Z）的 ASCII 控制代码。用户也可以使用其它键产生这些控制代码，下表是产生控制代码的键的 WM_CHAR 消息中的 wParam 值：

键	ASCII 代码	产生方法
Backspace	08H	Ctrl-H
Tab	09H	Ctrl-I
Ctrl-Enter	0AH	Ctrl-J
Enter	0DH	Ctrl-M
Esc	1BH	Ctrl-[

有时 Windows 程序将 Ctrl 与字母键的组合用作菜单加速键，此时，字母键不被转换成字符消息。

5.4.1 WM_CHAR 消息

当 Windows 程序需要处理来自键盘的字符时（例如，在字处理或者通信程序中），它就要处理 WM_CHAR 消息。你可能要对 Backspace 、 Tab 或者 Enter 键（也许还有 Linefeed 键）作一些特殊处理，不过，对所有其它字符处理应该不变：

```
case WM_CHAR :

     switch (wParam)
          {
          case '\b' :    // backspace
               [other program lines]
               break ;

          case '\t' :    // tab
               [other program lines]
```

```
            break ;

        case '\n' :     // linefeed
            [other program lines]
            break ;

        case '\r' :     // carriage return
            [other program lines]
            break ;

        default :       // character code
            [other program lines]
            break ;
        }
    return 0 ;
```

这段程序几乎与常规 MS-DOS 程序中的键盘字符处理完全相同。

5.4.2 死字符消息

Windows 程序经常忽略 WM_DEADCHAR 和 WM_SYSDEADCHAR 消息。在某些非 U.S.键盘上，有些键用于给字母加上音调。因为它们本身不产生字符，所以被称之为“死键”。例如，当安装德国键盘时，对于 U.S.键盘上的+/=键，德国键盘的对应位置就是一个死键，当未按下 Shift 键时它用于标识锐音，当按下 Shift 键时它用于标识抑音。

当用户按下这个死键时，窗口过程接收到一条 wParam 等于音调本身的 ASCII 代码的 WM_DEADCHAR 消息，然后用户再按字母键（例如 A 键），窗口过程接收到其 wParam 等于带有音调字母的 ASCII 代码的 WM_CHAR 消息。因此，用户程序不需要处理 WM_DEADCHAR 消息，原因是 WM_CHAR 消息已含有程序所需要的所有信息。Windows 甚至还构造了内部错误处理。如果在死键之后跟有不能带音调符号的字母（如字母 s），那么窗口过程将接收到两条 WM_CHAR 消息——前一条消息的 wParam 等于音调符号本身的 ASCII 代码(同传递到 WM_DEADCHAR 消息的 wParam 值相同)，另一条消息的 wParam 等于字母 s 的 ASCII 代码。

5.5 观察键盘消息

如果你希望知道 Windows 是怎样把键盘消息发送给程序的，那么可以阅读 KEYLOOK 程序（如图 5-3 中所示）。此程序在其客户区显示 Windows 发送给窗口过程的八个不同键盘消息的所有信息。

KEYLOOK.MAK

```
#-----------------------
# KEYLOOK.MAK make file
#-----------------------
```

```
keylook.exe : keylook.obj
     $(LINKER) $(GUIFLAGS) -OUT:keylook.exe keylook.obj $(GUILIBS)

keylook.obj : keylook.c
     $(CC) $(CFLAGS) keylook.c
```

KEYLOOK.C

```
/*-------------------------------------------------------
   KEYLOOK.C -- Displays Keyboard and Character Messages
                (c) Charles Petzold, 1996
  -------------------------------------------------------*/

#include <windows.h>
#include <stdio.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

RECT rect ;
int  cxChar, cyChar ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "KeyLook" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Keyboard Message Looker",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
```

```
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void ShowKey (HWND hwnd, int iType, char *szMessage,
              WPARAM wParam, LPARAM lParam)
     {
     static char *szFormat[2] =
                    "%-14s %3d    %c %6u %4d %3s %3s %4s %4s",
                    "%-14s    %3d %c %6u %4d %3s %3s %4s %4s" } ;
     char        szBuffer[80] ;
     HDC         hdc ;

     ScrollWindow (hwnd, 0, -cyChar, &rect, &rect) ;
     hdc = GetDC (hwnd) ;

     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

     TextOut (hdc, cxChar, rect.bottom - cyChar, szBuffer,
              wsprintf (szBuffer, szFormat [iType],
                   szMessage, wParam,
                   (BYTE) (iType ? wParam : ' '),
                   LOWORD (lParam),
                   HIWORD (lParam) & 0xFF,
                   (PSTR) (0x01000000 & lParam ? "Yes"  : "No"),
                   (PSTR) (0x20000000 & lParam ? "Yes"  : "No"),
                   (PSTR) (0x40000000 & lParam ? "Down" : "Up"),
                   (PSTR) (0x80000000 & lParam ? "Up"   : "Down"))) ;

     ReleaseDC (hwnd, hdc) ;
     ValidateRect (hwnd, NULL) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szTop[] =
               "Message        Key Char Repeat Scan Ext ALT Prev Tran" ;
     static char szUnd[] =
               "_______        ___ ____ ______ ____ ___ ___ ____ ____" ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

               GetTextMetrics (hdc, &tm) ;
```

```
          cxChar = tm.tmAveCharWidth ;
          cyChar = tm.tmHeight ;

          ReleaseDC (hwnd, hdc) ;

          rect.top = 3 * cyChar / 2 ;
          return 0 ;

     case WM_SIZE :
          rect.right  = LOWORD (lParam) ;
          rect.bottom = HIWORD (lParam) ;
          UpdateWindow (hwnd) ;
          return 0 ;

     case WM_PAINT :
          InvalidateRect (hwnd, NULL, TRUE) ;
          hdc = BeginPaint (hwnd, &ps) ;
          SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
          SetBkMode (hdc, TRANSPARENT) ;
          TextOut (hdc, cxChar, cyChar / 2, szTop, (sizeof szTop) - 1) ;
          TextOut (hdc, cxChar, cyChar / 2, szUnd, (sizeof szUnd) - 1) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_KEYDOWN :
          ShowKey (hwnd, 0, "WM_KEYDOWN", wParam, lParam) ;
          return 0 ;

     case WM_KEYUP :
          ShowKey (hwnd, 0, "WM_KEYUP", wParam, lParam) ;
          return 0 ;

     case WM_CHAR :
          ShowKey (hwnd, 1, "WM_CHAR", wParam, lParam) ;
          return 0 ;

     case WM_DEADCHAR :
          ShowKey (hwnd, 1, "WM_DEADCHAR", wParam, lParam) ;
          return 0 ;

     case WM_SYSKEYDOWN :
          ShowKey (hwnd, 0, "WM_SYSKEYDOWN", wParam, lParam) ;
          break ;        // ie, call DefWindowProc

     case WM_SYSKEYUP :
          ShowKey (hwnd, 0, "WM_SYSKEYUP", wParam, lParam) ;
          break ;        // ie, call DefWindowProc

     case WM_SYSCHAR :
          ShowKey (hwnd, 1, "WM_SYSCHAR", wParam, lParam) ;
          break ;        // ie, call DefWindowProc

     case WM_SYSDEADCHAR :
          ShowKey (hwnd, 1, "WM_SYSDEADCHAR", wParam, lParam) ;
          break ;        // ie, call DefWindowProc
```

```
        case WM_DESTROY :
             PostQuitMessage (0) ;
             return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 5-3 KEYLOOK程序

KEYLOOK 使用一种类似老式电传输出设备的显示方法。当 KEYLOOK 收到一个击键消息时，它调用 ScrollWindow 滚动整个客户区的内容，从而使其内容上滚一个字符高度。TextOut 用于从距离底部一个字符高度的位置开始显示一个新的信息行，这就如同电传输出一样简单。图 5-4 显示了当你输入单词 “ Windows ” 时屏幕所显示的内容。第一栏是键盘消息，第二栏是击键消息的虚拟键码，第三栏是字符消息的字符代码，其余六栏是 lParam 消息参数的六个域的状态。

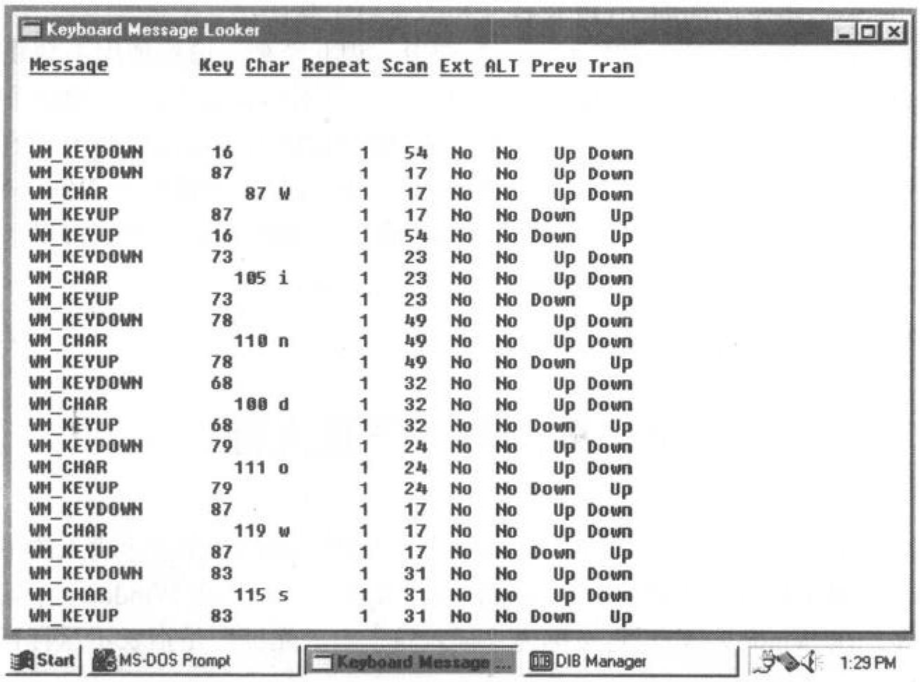


图 5-4 KEYLOOK的显示

KEYLOOK.C 主要使用了前面几个 SYSMETS 程序中已经出现过的 Windows 特性，不过，它还使用了几个新的函数。注意 KEYLOOK 的按列显示格式用缺省比例字体来实现是比较困难的，因为显示在每一行上的代码需要分为九个部分，以使各个部分都对齐。对这样的工作，采用固定间距的字体要容易得多。正如前一章所说的，这要用到我放在一条语句中的两个函数：

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT) ;
```

KEYLOOK 只要得到一个设备环境，就调用这两个函数。有三个地方会出现这种情况：ShowKey 函数，在 WndProc 中处理 WM_CREATE 消息时，以及处理 WM_PAINT 消息时。GetStockObject 函数获取指向现有图形对象的句柄，它就是在 3.0 以前版本的 Windows 中

使用的固定间距字体， SelectObject 调用将此对象选入设备环境中。此后，显示的所有文本都将使用固定间距的字体。通过调用：

```
SelectObject (hdc, GetStockObject (SYSTEM_FONT)) ;
```

可以切换回比例字体。

ShowKey 函数在显示新行之前调用 ScrollWindow 滚动前面的行。通常，这会导致窗口的一部分变为无效，从而产生 WM_PAINT 消息。 ShowKey 函数为了防止这种情况，就在其最后调用 ValidateRect 。

KEYLOOK 不保存它收到的击键，因而，在收到 WM_PAINT 消息时，它不能重新创建窗口。由于这个原因， KEYLOOK 在处理 WM_PAINT 消息期间仅仅在客户区域的上端简单地显示其标题。在处理 WM_PAINT 期间调用 BeginPaint 之前， KEYLOOK 先使整个窗口失效，从而允许整个窗口而不仅仅是无效区域被清除。

（ KEYLOOK 不保存击键，从而在处理 WM_PAINT 消息期间不能重新绘制窗口，这显然是一个缺陷，本章后面的 TYPER 程序将改正这个缺陷。）

KEYLOOK 在客户区的上方绘制一个标题以标识九个列。虽然使用下划线字体是可行的，但是这里使用了一个稍微不同的方法。我定义了分别称为 szTop（文本）和 szUnd（下划线）的两个字符串变量，并且在 WM_PAINT 期间将它们显示在窗口上方的同一位置。通常， Windows 使用“不透明”的方式显示文本，即在显示字符时， Windows 将擦除字符的背景区域，这将会导致第二个字符串（ szUnd ）擦除掉了第一个字符串（ szTop ）。为了防止这一点，可以将设备环境切换到“透明”方式：

```
SetMode (hdc, TRANSPARENT) ;
```

5.6 插入符（不是光标）

当你往一个程序中输入文本时，通常有一个下划线或者方框指示输入的下一个字符将出现在屏幕的哪个位置。这个标志通常被称为“光标”，但是在 Windows 下编程你必须改变这个习惯。在 Windows 中，它被称为“插入符”。“光标”是指表示鼠标位置的那个位图图象。

5.6.1 插入符函数

主要有五个插入符函数：

- CreateCaret ——创建与一个窗口有关的插入符
- SetCaretPos ——在窗口中设置插入符的位置
- ShowCaret ——显示插入符
- HideCaret ——隐藏插入符
- DestroyCaret ——撤消插入符

另外还有获取插入符位置（ GetCaretPos ）和获取以及设置插入符闪烁时间

（GetCaretBlinkTime 和 SetCaretBlinkTime）的函数。

习惯上，插入符是一条水平线、一条垂直线或者一个字符大小的方框。由于采用比例字体时字符的大小不固定，当你使用它们时，水平线或者方框不能设置为字符的大小。

你不能简单地在 WM_CREATE 消息期间创建一个插入符，在 WM_DESTORY 期间撤销一个插入符。插入符是一个“系统资源”，这意味着在系统中只有一个插入符。实际上，当程序在它的窗口中需要插入符时，它需要从系统“借用”插入符。

这种限制是不是太严格了？实际上并非如此，因为只有当一个窗口有输入焦点时才有必要在窗口中显示插入符。如果拥有插入符，则表示你可以输入文本。因为一个窗口仅有一个输入焦点，所以在整个系统中只需要一个插入符。

程序通过处理 WM_SETFOCUS 和 WM_KILLFOCUS 消息来决定是否拥有输入焦点。当一个窗口过程获得输入焦点时就会收到 WM_SETFOCUS 消息，当失去输入焦点时就会收到 WM_KILLFOCUS 消息。这些消息总是成对出现的：一个窗口过程总是在收到 WM_KILLFOCUS 之前接收到 WM_SETFOCUS 消息。在窗口的生命期内，它总是收到相同个数的 WM_SETFOCUS 和 WM_KILLFOCUS 消息。

使用插入符的主要规则很简单：在 WM_SETFOCUS 期间，窗口过程调用 CreateCaret；在 WM_KILLFOCUS 期间，窗口过程调用 DestroyCaret。

另外还有几条规则：插入符刚创建时是隐蔽的。如果想使插入符可见，那么你在调用 CreateCaret 之后，窗口过程还必须调用 ShowCaret。再者，当一个窗口过程处理一条非 WM_PAINT 消息而且希望在窗口内绘制某些东西时，它必须调用 HideCaret 隐藏插入符；在绘制完毕之后，再调用 ShowCaret 显示插入符。HideCaret 的影响具有累积效应，如果多次调用 HideCaret 而不调用 ShowCaret，那么只有调用相同次数的 ShowCaret，才能看到插入符。

5.6.2 TYPER 程序

图 5-5 中所示的 TYPER 程序使用了本章讨论的所有内容，你可以认为 TYPER 是一个特别简单的文本编辑器。在窗口中，你可以输入字符，使用光标移动键移动光标（或者移动插入符），使用 Escape 键擦除窗口的内容等，当你缩放窗口时也会擦除窗口的内容。但是本程序没有滚动、查找、替换、保存文件和拼写检查等功能。

为了简单起见，TYPER 程序使用一种固定间距的字体，因为编写一个处理比例字体的文本编辑器要困难得多。程序可以在好几个地方获取设备环境：在 WM_CREATE 消息期间，在 WM_KEYDOWN 期间，在 WM_CHAR 消息期间以及在 WM_PAINT 消息期间，每次都可以通过 GetStockObject 和 SelectObject 调用来选择固定间距字体。

在 WM_SIZE 期间，TYPER 计算窗口的字符宽度和高度并把值保存在 cxBuffer 和 cyBuffer 变量中，然后使用 malloc 分配缓冲区以保存在窗口内输入的所有字符，xCaret 和 yCaret 变量保存插入符位置。

在 WM_SETFOCUS 消息期间，TYPER 调用 CreateCaret 创建与字符有相同宽度与高度的插入符，SetCaretPos 设置插入符的位置，ShowCaret 使插入符可见。在 WM_KILLFOCUS 消息期间，TYPER 调用 HideCaret 和 DestroyCaret。

TYPER.MAK

```
#--------------------
# TYPER.MAK make file
#--------------------

typer.exe : typer.obj
     $(LINKER) $(GUIFLAGS) -OUT:typer.exe typer.obj $(GUILIBS)

typer.obj : typer.c
     $(CC) $(CFLAGS) typer.c
```

TYPER.C

```
/*------------------------------------
   TYPER.C -- Typing Program
              (c) Charles Petzold, 1996
  ------------------------------------*/

#include <stdlib.h>

#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Typer" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Typing Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
```

```
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char *pBuffer = NULL ;
     static int  cxChar, cyChar, cxClient, cyClient, cxBuffer,
                 cyBuffer, xCaret, yCaret ;
     HDC         hdc ;
     int         x, y, i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;

               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight ;

               ReleaseDC (hwnd, hdc) ;
               return 0 ;

          case WM_SIZE :
                              // obtain window size in pixels

               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

                              // calculate window size in characters

               cxBuffer = max (1, cxClient / cxChar) ;
               cyBuffer = max (1, cyClient / cyChar) ;

                              // allocate memory for buffer and clear it

               if (pBuffer != NULL)
                    free (pBuffer) ;

               if ((pBuffer = (char *) malloc (cxBuffer * cyBuffer))
                   == NULL)
                    MessageBox (hwnd, "Window too large.  Cannot "
                                      "allocate enough memory.",
                                      "Typer",
                                      MB_ICONEXCLAMATION | MB_OK) ;
               else
```

```
               for (y = 0 ; y < cyBuffer ; y++)
                    for (x = 0 ; x < cxBuffer ; x++)
                         BUFFER(x,y) = ' ' ;

                         // set caret to upper left corner
          xCaret = 0 ;
          yCaret = 0 ;

          if (hwnd == GetFocus ())
               SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;

          return 0 ;

     case WM_SETFOCUS :
                         // create and show the caret

          CreateCaret (hwnd, NULL, cxChar, cyChar) ;
          SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
          ShowCaret (hwnd) ;
          return 0 ;

     case WM_KILLFOCUS :
                         // hide and destroy the caret
          HideCaret (hwnd) ;
          DestroyCaret () ;
          return 0 ;

     case WM_KEYDOWN :
          switch (wParam)
               {
               case VK_HOME :
                    xCaret = 0 ;
                    break ;

               case VK_END :
                    xCaret = cxBuffer - 1 ;
                    break ;

               case VK_PRIOR :
                    yCaret = 0 ;
                    break ;

               case VK_NEXT :
                    yCaret = cyBuffer - 1 ;
                    break ;
                case VK_LEFT :
                    xCaret = max (xCaret - 1, 0) ;
                    break ;

               case VK_RIGHT :
                    xCaret = min (xCaret + 1, cxBuffer - 1) ;
                    break ;

               case VK_UP :
                    yCaret = max (yCaret - 1, 0) ;
```

```
                    break ;

               case VK_DOWN :
                    yCaret = min (yCaret + 1, cyBuffer - 1) ;
                    break ;

               case VK_DELETE :
                    for (x = xCaret ; x < cxBuffer - 1 ; x++)
                         BUFFER (x, yCaret) = BUFFER (x + 1, yCaret) ;

                    BUFFER (cxBuffer - 1, yCaret) = ' ' ;

                    HideCaret (hwnd) ;
                    hdc = GetDC (hwnd) ;

                    SelectObject (hdc,
                         GetStockObject (SYSTEM_FIXED_FONT)) ;

                    TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
                             & BUFFER (xCaret, yCaret),
                             cxBuffer - xCaret) ;

                    ShowCaret (hwnd) ;
                    ReleaseDC (hwnd, hdc) ;
                    break ;
               }

          SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
          return 0 ;

     case WM_CHAR :
          for (i = 0 ; i < (int) LOWORD (lParam) ; i++)
               {
               switch (wParam)
                    {
                    case '\b' :                    // backspace
                         if (xCaret > 0)
                              {
                              xCaret-- ;
                              SendMessage (hwnd, WM_KEYDOWN,
                                           VK_DELETE, 1L) ;
                              }
                         break ;

                    case '\t' :                    // tab
                         do
                              {
                              SendMessage (hwnd, WM_CHAR, ' ', 1L) ;
                              }
                         while (xCaret % 8 != 0) ;
                         break ;

                    case '\n' :                    // line feed
                         if (++yCaret == cyBuffer)
                              yCaret = 0 ;
```

```
                    break ;

               case '\r' :                   // carriage return
                    xCaret = 0 ;

                    if (++yCaret == cyBuffer)
                         yCaret = 0 ;
                    break ;

               case '\x1B' :                 // escape
                    for (y = 0 ; y < cyBuffer ; y++)
                         for (x = 0 ; x < cxBuffer ; x++)
                              BUFFER (x, y) = ' ' ;

                    xCaret = 0 ;
                    yCaret = 0 ;

                    InvalidateRect (hwnd, NULL, FALSE) ;
                    break ;

               default :                     // character codes
                    BUFFER (xCaret, yCaret) = (char) wParam ;

                    HideCaret (hwnd) ;
                    hdc = GetDC (hwnd) ;

                    SelectObject (hdc,
                         GetStockObject (SYSTEM_FIXED_FONT)) ;

                    TextOut (hdc, xCaret * cxChar,
                              yCaret * cyChar,
                              & BUFFER (xCaret, yCaret), 1) ;

                    ShowCaret (hwnd) ;
                    ReleaseDC (hwnd, hdc) ;

                    if (++xCaret == cxBuffer)
                         {
                         xCaret = 0 ;

                         if (++yCaret == cyBuffer)
                              yCaret = 0 ;
                         }
                    break ;
               }
          }

     SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
     return 0 ;

case WM_PAINT :
     hdc = BeginPaint (hwnd, &ps) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

     for (y = 0 ; y < cyBuffer ; y++)
          TextOut (hdc, 0, y * cyChar, & BUFFER(0,y), cxBuffer) ;
```

```
            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 5-5 TYPER程序

处理 WM_KEYDOWN 和 WM_CHAR 消息的程序很大。对 WM_KEYDOWN 的处理大多要涉及光标移动键，另外， Home 和 End 把插入符送至一行的开始和末尾处， PageUp 和 PageDown 把插入符送至窗口的顶端和底部，箭头的用法不变。对 Delete 键， TYPER 将缓冲区中从插入符之后的那个位置开始到行尾的所有内容向前移动，并在行尾显示空格。

WM_CHAR 处理 Backspace 、 Tab 、 Linefeed(Ctrl-Enter)、 Enter 、 Escape 和字符键。注意在处理 WM_CHAR 消息时（假设用户输入的每个字符都非常重要），我使用了 lParam 中的重复记数；而在处理 WM_KEYDOWN 消息时却不这么作（禁止有害的重复滚动）。对 Backspace 和 Tab 的处理由于使用了 SendMessage 函数而得到简化， Backspace 由 Delete 来模拟，而 Tab 由若干个空格来模拟。

前面我已经提到过，在非 WM_PAINT 消息期间，如果要在窗口中绘制内容，则应该隐蔽光标。程序为 Delete 键处理 WM_KEYDOWN 消息和为字符键处理 WM_CHAR 消息时即是如此。在这两种情况下， TYPER 改变缓冲区中的内容，然后在窗口中绘制一个或者多个新字符。

我用 TYPER 输入一份演讲稿，如图 5-6 中所示。

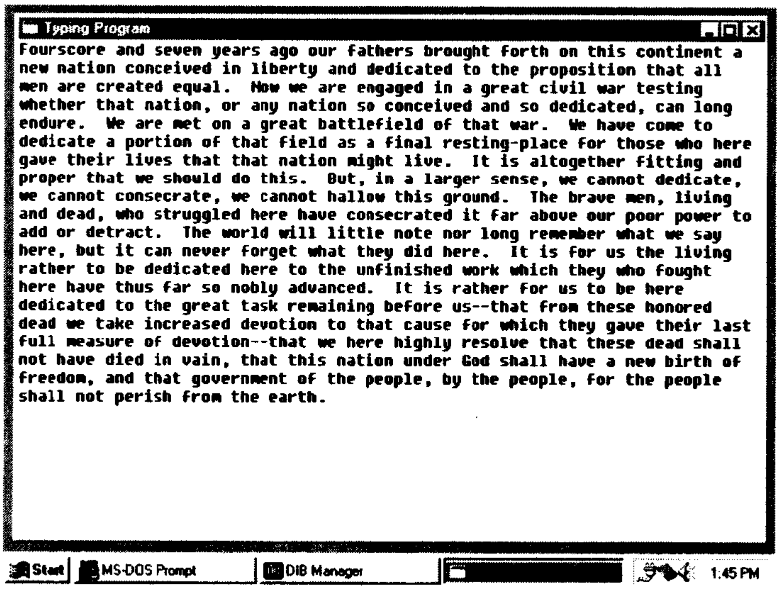


图 5-6 TYPER显示

5.7 Windows 字符集

我在前面已经提到过，带有死字符前缀的字母键会产生 WM_CHAR 消息，此时 wParam 是带音调字符的 ASCII 代码，这可能有些不好理解，因为 ASCII 字符集不包括任何带有音调字符的代码。此时 wParam 值的确切含义是什么呢？回答此问题要求我们先建立字符集的概念，这对以后关于字体的讨论是有用的，而且对键盘的处理也是非常重要的。

标准 7 位 ASCII 字符集定义 0 至 31 之间的代码（ 0X1F ）及代码 127 （ 0X7F ）为控制字符， 32 （ 0X20 ）至 126 （ 0X7E ）之间的代码为可显示字符，这些字符都没有音调。由于 PC 机使用 8 位编码，计算机制造商经常定义使用了全部 256 个代码的字符集，而非 128 字符的 ASCII 代码。增加的代码可以用于带音调的字符。“扩展字符集”包括 ASCII 字符集和多达 128 个的其它字符。

如果 Wndows 支持这样的扩展字符集，那么显示带音调的字符是很容易的，但是 Windows 不支持一个简单的扩展字符集。 Windows 支持两个扩展字符集，不幸的是，这两个字符集的存在并没有使我们感到双倍的方便。

5.7.1 OEM 字符集

首先，我们回顾一下 Windows 运行的硬件环境—— IBM PC 及其兼容机。在八十年代早期， IBM PC 的开发人员决定把ASCII 字符集扩展为图 5-7 所示的字符集， 0x20 到 0x7E 之间的代码是 ASCII 字符集中的显示字符，其余是非标准的——或者至少那时如此。

图 5-7 按字符代码排列的IBM扩展字符集

不可忽视这个字符集，它已经被固化在成千上万个 IBM 视频适配器、打印机、系统 BIOS 板上的 ROM 芯片中， IBM 兼容机和外设也用到了这些字符集，这个字符集是所谓的“ IBM 标准”的一部分。为 IBM PC 编写的许多非 Windows 的文本方式程序都要求这个扩展字符集，因为它们在其屏幕输出中用到块绘制和线绘制字符（代码从 B0H 至 DFH ）。

唯一问题是 IBM 扩展字符集不适宜于 Windows 使用。首先， PC 机文本模式下的应用程序所使用的块绘制和线绘制字符在 Windows 中是不需要的，因为 Windows 能真正做到图形化，绘制一条水平线比显示 0xC4 字符串还要方便。第二，希腊字母和数学符号在 Windows 中不如大多数欧洲语言中的重音字母重要，程序需要显示数学符号时，可以用图形函数进行绘制。

简单地说， Windows 支持 IBM 字符集，但是 Windows 把它放在次要地位——主要是为了以前的应用程序能在 Windows 下运行。 Windows 应用程序一般不使用 IBM 字符集。在 Windows 文档中， IBM 字符集是指“ OEM 字符集”， OEM 字符集的更准确含义是，当前运行 Windows 的那个机器上原有的字符集。

5.7.1.1 DOS下的国际支持

现在有许多 IBM PC 字符集的变体，称为“代码页”。美国和大多数欧洲国家使用的变体称为代码页 437 。挪威、丹麦、葡萄牙等国家出售的系统使用不同的特殊代码页，这些代码页含有更多的与其国家所使用的语言有关的特殊字符。最近，这些国家开始使用代码页 850 ，它含有少数几个图形符号、更多的音调字母和其它特殊字符。

Windows 通过安装对应于系统的代码页的 OEM 字体（在 Windows 中运行 DOS 应用程序和剪贴板浏览器都要用到）以及用于 CharToOem 和 OemToChar 函数（以后将加以讨论）的适当转换表来支持代码页。

Windows Setup 程序将根据当前系统配置中的地域设置，选择适当的代码页。

5.7.2 ANSI 字符集

Windows 和 Windows 程序在大多数情况下使用的扩展字符集称为“ ANSI 字符集”，实际上是一种 ISO 标准。当你的程序收到 WM_CHAR 消息时， wParam 参数是 ANSI 字符代码。 ANSI 字符如图 5-8 中所示，正如你所看到的，代码 0x20 到 0x7E 表示 OEM 字符集和 ASCII 字符集中出现的相同字符。用实心方块表示的字符为未定义字符。在其它输出设备（如打印机）上，它们可能以不同的形式显示。 TrueType 字体为 ANSI 代码 0x80 到 0x9F 定义了一些附加字符。

5.7.3 OEM 、 ANSI 和字体

Windows 为显示 ANSI 和 OEM 字符集而准备了不同的字体。当你获得一个指向设备环境的句柄时，设备环境的属性之一就是字体，缺省时使用 ANSI 字符集的 SYSTEM_FONT 或者说“系统字体”。如果你想显示 OEM 字符集中的字符，那么可以用以下代码在设备

环境中选择 OEM_FIXED_FONT（也称为终端字符）：

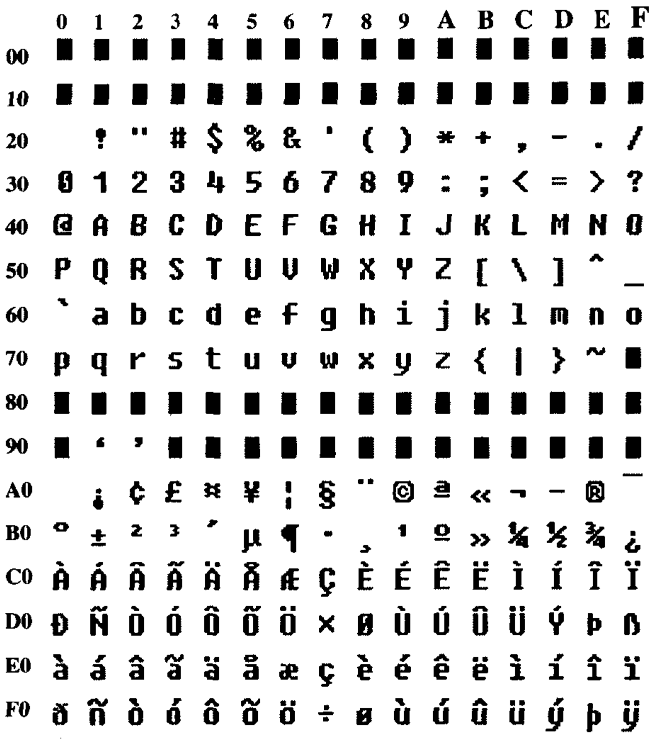


图 5-8 按字符代码排列的ANSI字符集

```
SelectObject (hdc, GetStockObject(OEM_FIXED_FONT)) ;
```

5.8 国际化考虑

现在说明为什么要在键盘这一章里讨论字体。我们已经提到过，当 Windows 用户在非 U.S.键盘上输入带音调的字符时，WM_CHAR 消息的 wParam 参数是 ANSI 字符集中该字符的代码。

因此，如果你需要在显示器上显示该字符，那么最好使用 ANSI 字符集的字体（如 SYSTEM_FONT 或者 SYSTEM_FIXED_FONT）。如果你用 OEM_FIXED_FONT 字体代替，那么写到显示器上的字符将是不正确的。当你要求程序能在欧洲市场上销售时，有几条简单的规则可以使 Windows 程序中的键盘逻辑满足该市场的要求。

5.8.1 字符集的处理

当你得到一个 WM_CHAR 消息时， wParam 的值大于 128 也可能是合法的，不要假设大于 127 的代码是非法字符。

你可能想把一个字符转换成大写格式，请不要使用自己的算法：

```
if (ch >= 'a' && ch<='z')
```

```
ch -= 32 ;          //错误！！！
```

即使是编写非 Windows C 程序，这也是一个很不好的习惯。但是也不要使用标准 C 函数：

```
ch = toupper (ch) ;          //错误！！！
```

这两个函数只能处理 ANSI 字符集的小写形式，它们不会把 0xE0 转换成 0xC0 。

你应该使用 Windows 函数 CharUpper 和 CharLower 。如果 pString 是一个以零终结的字符串，那么你可以使用 CharUpper 将它转换为大写形式：

```
CharUpper (pString) ;
```

或者对非零终结的字符串可以使用 CharUpperBuff 函数：

```
CharUpperBuff (pString, nLength) ;
```

你也可以使用 CharUpper 来转换一个单字符，但是需要作一些强制，因为参数的高位字节必须为 0：

```
ch = CharUpper ((PSTR) (LONG) (BYTE) ch) ;
```

如果ch定义为一个无符号字符，则不需要BYTE强制转换。为了转换成小写形式， Windows 还提供了 CharLower 和 CharLowerBuff 函数。

如果你确实需要编写能方便地转换到外国语言的 Windows 程序，那么你还应该研究一下 CharNext 和 CharPre 函数，这些函数实现多字节字符集的处理，常用于远东国家。这些字符集要求 256 个以上的字符，其中的一些要使用 2 个字节。如果你使用普通的 C 指针算法来扫描字符串（比如搜索目录路径中的反斜杠），你可能认为已经找到了该字符，而实际上却只是找到双字节代码的第二个字节。 CharNext 和 CharPre 使用指向字符串的远指针，返回一个经过正确增减后的远指针。

5.8.2 与 MS-DOS 交流

如果 Windows 是运行在机器上的唯一操作系统，那么你可以忘掉 OEM 字符集，而只使用 ANSI 字符集。不过，用户可以在 MS-DOS 环境中创建文件并在 Windows 中使用它；也可以在 Windows 中创建文件再回到 MS-DOS 中使用它。不幸的是， MS-DOS 使用 OEM 字符集。

下面是可能会发生交流问题的例子。假定一个说德语的 PC 机用户用 MS-DOS 程序，如 EDLIN ，创建了一个叫做ÜBUNGEN.TXT(实践训练)的文件。在 IBM PC 机上，Ü是 IBM（即 OEM ）字符集的一部分，其代码是 154 或者 0x9A （在 IBM PC 机上用 U.S.键盘运行 MS-DOS 时，通过数字键盘输入 Alt-154 也能产生这个字符）， MS-DOS 在文件目录中使用这个字符代码。

如果 Windows 程序使用 MS-DOS 函数调用来获得一个文件目录，然后用 ANSI 字符集

中的字体直接把目录写到显示器上，那么ÜBUNGEN 的第一个字母将以实心方块的形式出现，因为在 ANSI 字符集中代码 154 是一个未定义字符。Windows 程序需要把 IBM 扩展字符集代码 154 （0x9A）转换为 ANSI 字符集代码 220 （或者 0xDC），即字母Ü在 ANSI 字符集中的代码。Windows OemToChar 函数可以做到这一点，它接受指向字符串的两个远指针，把第一个字符串的 OEM 字符转换为 ANSI 字符并存在第二个字符串中：

```
OemToChar (lpszOemStr, lpszAnsiStr) ;
```

现在让我们来看一个相反的例子。说德语的用户用 Windows 程序创建一个称为“ÜBUNGEN.TXT ”的文件，文件名的第一个字符的代码是 220 （0xDC）。如果你用 MS-DOS 函数调用打开此文件，MS-DOS 将使用文件名中的该字符。当用户在 MS-DOS 上寻找该文件时，第一个字符显示为一个方块。在使用 MS-DOS 函数调用之前，你必须把文件名转换为 OEM 字符集：

```
CharToOem (lpszAnsiStr, lpszOemStr) ;
```

它将 220 （0xDC）转换为 154 （0x9A）。Windows 还包括两个不要求以零终结的字符串函数，称为 CharToOemBuff 和 OemToCharBuff。

Windows 的 OpenFile 调用可以实现字符转换，如果你使用它，那么就不必自己进行转换。如果你使用 MS-DOS 函数调用来获取文件名列表（如 Windows 的文件管理器所做的），那么这些文件名应该在显示之前被传递给 OemToChar。

当文件既在 Windows 中使用，又在 MS-DOS 中使用时，还会产生文件内容相互转换的问题。如果你的 Windows 程序使用一个 MS-DOS 程序创建的文件，那么必须把文件的内容传递给 OemToChar 函数。同样，如果你的 Windows 程序创建一个将在 MS-DOS 程序中使用的文件时，那么你需要使用 CharToOem 来转换文本。

OemToChar 和 CharToOem 函数在键盘驱动程序中，它们包含了非常简单的查表动作。OemToChar 例程把 0x80 至 0xFF 的 OEM 代码转换成同 OEM 字符最相似的 ANSI 字符集中的代码。有时，这种转换只能做到大体相似。例如，IBM 字符集中的大多数画线字符被转换为加号、破折号和垂直线。0x00 至 0x1F 之间的大多数 OEM 字符不被转换为 ANSI 代码。

CharToOem 例程把 0xA0 至 0xFF 之间的 ANSI 代码转换为 OEM 字符集中的代码。当 ANSI 字符集中带音调字符在 OEM 字符集中没有相对应的字符时，就转换为正常的 ASCII 代码，而不再带有音调。

5.8.3 数字键盘的使用

众所周知，IBM PC 键盘和 BIOS 允许你输入 IBM 扩展字符集代码，方法是按住 Alt 键，在数字小键盘上输入表示 OEM 字符的三位十进制代码，然后再释放 Alt 键。在 Windows 中有两种方法可以做到这一点。

第一，当你使用数字小键盘输入 Alt-[OEM 代码]时，Windows 给出与 OEM 字符（由

OEM 代码表示）最接近的 ANSI 字符代码（在 WM_CHAR 消息中的 wParam 参数中）。也就是说，在产生 WM_CHAR 消息之前，Windows 先把此代码送给 OemToChar 函数，这样做对用户来说是非常方便的。如果你没有外语键盘，而又习惯于用 Alt-154 的方法来输入ü，可以在 Windows 程序中做到这一点，从而就不用再去学习 ANSI 字符代码了。

第二，如果你想用 U.S.键盘产生 ANSI 扩展字符码，那么你只要在数字小键盘上输入 Alt-0[OEM 代码]即可。WM_CHAR 消息中的 wParam 参数就是该 OEM 代码。从而，Alt-0220 也是ü，你可以运行 KEYLOOK 或者 TYPERR 程序来进行一些尝试。

5.8.4 Windows NT 的 Unicode 解决方案

对于面向国际市场开发应用系统的程序开发人员来说，他们不得不去应付针对纯 7 位 ASCII 缺陷的各种很不充分的解决方案，其中包括诸如代码页和双字节字符集等。我们迫切地需要一种更好的解决方案，目前为止，Unicode 可能就是这样的一种方案。

Unicode 是一种字符编码，它为每个字符使用一个统一的 16 位码。每一种可能在计算机通信中使用的世界语言中的每个字符都在 Unicode 中有一种表示，这些语言包括汉字的象形符号、日文和韩文。Unicode 是由一个计算机工业组织（包括所有的大公司）组成的研究机构创建的，并在由 Addison-Wesley 发表的著作 The Unicode Standard（Unicode 标准）中进行了归纳整理。

不幸的是，Windows 95 只有对 Unicode 的基本支持，并且它无法通过键盘驱动来使用 Unicode 字符。在这方面，它和 Windows NT 很不同，Windows NT 提供了对 Unicode 的完全支持。

显然，使程序（以及程序员的思想）适应 16 位字符的潮流是一项相当重要的工作，但是如果我们计划在个人计算机的屏幕和打印机上显示世界上的所有语言，那么这应该是一项值得做的事。如果你对于在 Windows NT 下实现的 Unicode 的概念和技术感兴趣，那么你可以找出我在 1993 年 10 月 26 日、11 月 9 日、11 月 23 日和 12 月 7 日的 PC Magazine 上的“Environment”栏目来看一看。

第 6 章 鼠　　标

鼠标是有一个或多个键的定位设备。虽然也可以使用诸如触摸屏和光笔之类的输入设备，但是只有鼠标（以及常用在膝上计算机上的跟踪球等）才是渗透了 PC 市场的唯一设备。

情况并非总是如此。当然， Windows 的早期开发人员认为他们不应该要求用户为了运行其产品而必须买只鼠标。因此，他们将鼠标作为一种可选的附加设备，而为 Windows 中的所有操作以及图标提供一种键盘接口。第三方开发人员也被鼓励使用一种键盘接口来模拟鼠标的功能，并且这本书以前的版本也是这么做的。

虽然鼠标几乎统治了 Windows 机器，但是我仍然相信这一原则。打字员尤其喜欢让他们的手保持在键盘上，并且我认为每个人都有在杂乱的桌上找不到鼠标的经历。由于这个原因，我仍然建议——只要可能，你就应该增加一个键盘接口来模拟鼠标的功能。

6.1 鼠 标 基 础

Windows 95 能支持单键、双键或者三键鼠标，也可以使用操纵杆或者光笔来模拟单键鼠标。因为单键鼠标是最小公共部分，因此许多 Windows 程序员总是避免使用双键或三键鼠标。不过，由于双键鼠标已经成为事实上的标准，因而不使用第二个键的传统已经不再合理了。当然，第二个鼠标键是用于激活一个“环境菜单”（即出现在普通菜单条之外的窗口中的菜单）或者用于特殊的拖动操作（拖动将在后面加以解释）。

你可以用 GetSystemMetrics 函数来判定鼠标是否存在：

```
fMouse = GetSystemMetrics (SM_MOUSEPRESENT) ;
```

如果已经安装了鼠标，那么 fMouse 的值为 TRUE （非零）。为了确定所安装鼠标上键的个数，使用：

```
cButton = GetSystermMetrics (SM_CMOUSEBUTTONS) ;
```

如果没有安装一个鼠标，那么函数将返回 0 。

左撇子用户可以使用 Windows 控制面板来切换鼠标键。虽然一个应用程序可以通过在 GetSystermMetrics 中使用 SM_SWAP_BUTTON 参数来确定是否进行了这种切换，但通常没有这个必要。由食指触发的键被认为是左键，即使事实上是位于鼠标的右边。不过，在一个培训用的程序中，你可能想在屏幕上画一个鼠标，在这种情况下，你可能想知道鼠标键是否被切换过了。

6.1.1 一些简单的定义

当 Windows 用户移动鼠标时， Windows 在显示屏上移动一个称为“鼠标光标”的小位图。鼠标光标有一个指向显示屏上精确位置的单象素的“热点”。

显示驱动程序含有几个预定义的可以由应用程序使用的鼠标光标。最常用的是在 Windows 头文件中定义的称为 IDC_ARROW 的斜箭头，热点在箭头的顶端。 IDC_CROSS 光标（在本章中的 BLOKOUT 程序中用到）的热点在十字交叉线的中心。 IDC_WAIT 光标是一个时钟，通常用于指示程序正在运行。程序员也可以设计自己的光标（第 9 章中我们就是这样作的）。当定义窗口类结构时指定特定窗口的缺省光标，例如：

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW) ;
```

下面是一些描述鼠标键动作的术语：

- 单击鼠标键——按下并放开一个鼠标键
- 双击鼠标键——快速按下并放开鼠标键两次
- 拖动——按住鼠标键并移动鼠标

对三键鼠标，三个键分别称为左键、中键、右键。在 Windows 头文件中定义的与鼠标有关的标识符使用缩写 LBUTTON 、 MBUTTON 和 RBUTTON 。双键鼠标只有左键与右键，单键鼠标只有一个左键。

6.2 客户区鼠标消息

在前一章中你已经看到， Windows 只能把键盘消息发送给拥有输入焦点的窗口。鼠标消息与此不同：只要鼠标跨越窗口或者在某窗口中按下鼠标键，那么窗口过程就会收到鼠标消息，而不管该窗口是否活动或者是否拥有输入焦点。 Windows 为鼠标定义了 21 种消息，不过，其中有 11 个消息同客户区无关（下面称之为“非客户区”消息）， Windows 程序经常忽略这些消息。

当鼠标移过窗口的客户区时，窗口过程收到 WM_MOUSEMOVE 消息。当在窗口的客户区中按下或者释放一个鼠标键时，窗口过程会接收到下面这些消息：

键	按下	释放	双击键
左	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
中	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
右	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

只有对三键鼠标，窗口过程才会接收到“ MBUTTON ”消息；只有对双键或者三键鼠标，才会接收到“ RBUTTON ”消息。仅当定义的窗口类能接收“ DBLCLK ”消息之后，

窗口过程才能接收到这些消息（下面将加以说明）。

对于所有这些消息来说，其 lParam 值均含有鼠标的位置，低位字为 x 坐标，高位字为 y 坐标，这两个坐标是相对于窗口客户区左上角的位置。Windows 头文件中的 LOWORD 和 HIWORD 宏可以从 lParam 中抽取 x、y 坐标。wParam 的值指示鼠标键以及 Ctrl 和 Shift 键的状态。你可以使用头文件中定义的位屏蔽来测试 wParam。MK 前缀代表"鼠标键"。

MK_LBUTTON	按下左键
MK_MBUTTON	按下中键
MK_RBUTTON	按下右键
MK_SHIFT	按下 Shift 键
MK_CONTROL	按下 Ctrl 键

当你把鼠标移过窗口的客户区时，Windows 并不为鼠标的每个可能的象素位置都产生一条 WM_MOUSEMOVE 消息。你的程序接收到 WM_MOUSEMOVE 消息的次数依赖于鼠标硬件以及你的窗口过程处理鼠标移动消息的速度。当你运行下面将描述的 CONNECT 程序时，你将会更好地理解 WM_MOUSEMOVE 消息的速率。

如果你在非活动窗口的客户区中按下鼠标左键，Windows 将把活动窗口改为在其中按下鼠标键的窗口，然后把 WM_LBUTTONDOWN 消息送到该窗口过程。当窗口过程得到 WM_LBUTTONDOWN 消息时，你的程序就可以安全地假定该窗口是活动的了。不过，你的窗口过程可能在未接收到 WM_LBUTTONDOWN 消息的情况下先接收到了 WM_LBUTTONUP 的消息。如果在一个窗口中按下鼠标键，然后移动到用户窗口并释放它，就会出现这种情况。类似地，当鼠标键在另一个窗口中被释放时，窗口过程只能接收到 WM_BUTTONDOWN 消息而没有相应的 WM_LBUTTONUP 消息。

这些规则有两个例外：

- 窗口过程可以“捕获鼠标”并且连续地接收鼠标消息，即使此时鼠标在该窗口客户区之外。你将在本章的后面学习如何捕获鼠标。
- 如果正在显示一个模态系统消息框或者模态系统对话框，那么其它程序不能接收鼠标消息。当模态系统消息框或者对话框活动时，禁止切换到其它窗口或者程序（当你关闭程序时，将会显示一个模态系统消息框的例子）。

6.2.1 简单的鼠标处理：一个例子

图 6-1 中所示的 CONNECT 程序能作一些简单的鼠标处理，使你对 Windows 如何向你的程序发送鼠标消息有一些体会。

CONNECT.MAK

```
#-----------------------
# CONNECT.MAK make file
#-----------------------
```

```
connect.exe : connect.obj
     $(LINKER) $(GUIFLAGS) -OUT:connect.exe connect.obj $(GUILIBS)

connect.obj : connect.c
     $(CC) $(CFLAGS) connect.c
```

CONNECT.C

```
/*--------------------------------------------------
   CONNECT.C -- Connect-the-Dots Mouse Demo Program
                (c) Charles Petzold, 1996
  --------------------------------------------------*/

#include <windows.h>

#define MAXPOINTS 1000
#define MoveTo(hdc, x, y) MoveToEx (hdc, x, y, NULL)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Connect" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Connect-the-Points Mouse Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
```

```
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static POINT points[MAXPOINTS] ;
     static int   iCount ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     int          i, j ;

     switch (iMsg)
          {
          case WM_LBUTTONDOWN :
               iCount = 0 ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case WM_MOUSEMOVE :
               if (wParam & MK_LBUTTON && iCount < 1000)
                    {
                    points[iCount  ].x = LOWORD (lParam) ;
                    points[iCount++].y = HIWORD (lParam) ;

                    hdc = GetDC (hwnd) ;
                    SetPixel (hdc, LOWORD (lParam), HIWORD (lParam), 0L) ;
                    ReleaseDC (hwnd, hdc) ;
                    }
               return 0 ;

          case WM_LBUTTONUP :
               InvalidateRect (hwnd, NULL, FALSE) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;

               for (i = 0 ; i < iCount - 1 ; i++)
                    for (j = i + 1 ; j < iCount ; j++)
                         {
                         MoveTo (hdc, points[i].x, points[i].y) ;
                         LineTo (hdc, points[j].x, points[j].y) ;
                         }

               ShowCursor (FALSE) ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;
```

```
        case WM_DESTROY :
             PostQuitMessage (0) ;
             return 0 ;
        }
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
   }
```

图 6-1 CONNECT程序

CONNECT 处理三条鼠标消息：

- WM_LBUTTONDOWN —— CONNECT 清除客户区。
- WM_MOUSEMOVE —— 如果按下左键，那么 CONNECT 就在客户区中的鼠标位置处绘制一个黑点。
- WM_LBUTTONUP —— CONNECT 把客户区中绘制的黑点同其它每个黑点连接起来。有时会产生一个漂亮的图形，而有时则会是黑乎乎的一团糟（见图 6-2）。

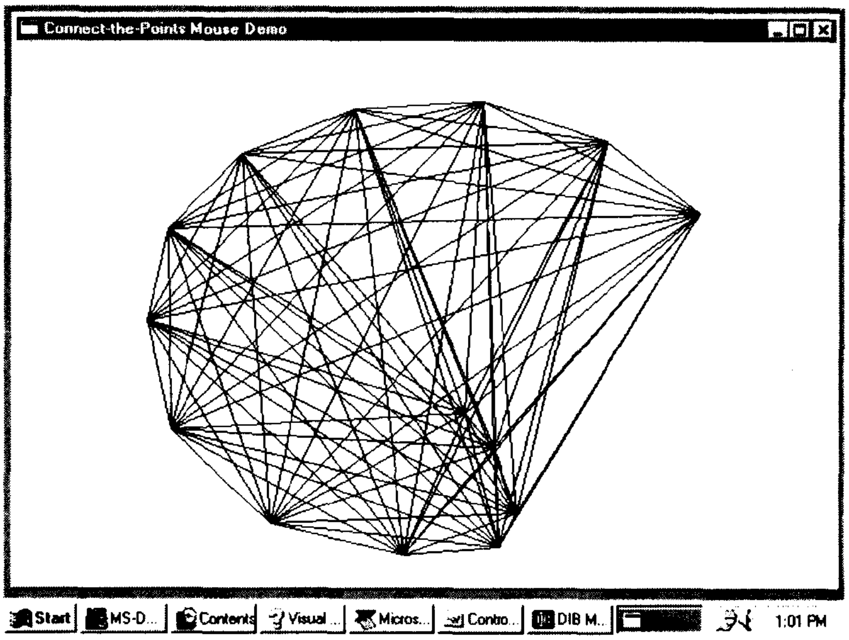


图 6-2 CONNECT显示

使用 CONNECT 的方法：把鼠标光标移动到客户区中，按下左键，移动一下位置，释放左键。对几个构成曲线的点， CONNECT 能处理得很好，方法是按住左键，快速移动鼠标，这样就可以绘制出该曲线图案。 CONNECT 使用了几个简单的图形设备接口（ GDI ）函数。 SetPixel 用指定颜色绘制出象素——本例中为黑色（对高分辨率的显示器，象素几乎看不见）。绘制直线需要两个函数： MoveTo 标出直线的起始坐标， LineTo 绘出直线（注意我将 MoveTo 定义为一个使用 MoveToEx 的宏）。

如果你在释放鼠标键之前把鼠标光标移到客户区之外，那么 CONNECT 就不会连接这些点，因为它没有收到 WM_LBUTTONUP 消息。如果你把鼠标移回客户区内并按下左键，那么 CONNECT 将清除客户区（如果想在客户区外释放左键后还继续进行画图，那么可以在客户区外按下鼠标再移回客户区中）。

CONNECT 最多可以保存 1000 个点，它绘制的线数等于：

$$\frac{(P)\times(P-1)}{2}$$

其中 P 为点的个数。如果有 1000 个点，则要绘制 500,000 条直线，大约需要几分钟才能画完。由于 Windows 95 是一种有优先级的多任务环境，因此你可以在这一段时间切换到别的程序中。但是，当程序正在忙的时候，你将无法对 CONNECT 程序作任何事（诸如移动或者缩放等）。在第 14 章中，我们将讨论解决这一问题的方法。

因为 CONNECT 可能会花一些时间来绘制直线，因此在处理 WM_PAINT 消息时它将切换到时钟光标，然后再恢复原状。这要求使用两个现有光标来调用 SetCursor 。CONNECT 还调用两次 ShowCursor ，一次用 TRUE 参数，另一次用 FALSE 参数。我将在这一章的后面，即“使用键盘模拟鼠标”这一节中更详细地讨论这些调用。

如果 CONNECT 正忙于绘制直线，你可以按住鼠标键，移动并释放它，此时将不会产生任何事件。由于 CONNECT 正在忙碌而不会进行任何 GetMessage 调用，因此它不接收这些消息。而在完成了绘制之后， CONNECT 也不能接收这些消息，因为此时鼠标键已经被释放了。在这一方面，鼠标同键盘不一样。 Windows 会处理每一个击键，似乎是认为它们比较重要。而如果在客户区中按下并释放鼠标键时程序正在运行，那么将放弃对鼠标键的处理。

现在，试试这样做：当 CONNECT 正忙于运行一个冗长的绘制过程时，按住鼠标键并四处移动它。在 CONNECT 完成绘制之后，它将会从队列中取得 WM_LBUTTONDOWN 消息（并清除客户区），因为鼠标键现在仍然被按着。但是它只能接收到在收到这个 WM_LBUTTONDOWN 消息之后才发生的 WM_MOUSEMOVE 消息。

有时，我们使用“跟踪”这个词代表程序处理鼠标移动的方法，但是，跟踪并不意味着程序在窗口过程中的某个循环里不断跟随鼠标在显示器上的运动。实际上，窗口过程处理每条鼠标消息，然后迅速退出。

6.2.2 处理换档键

当 CONNECT 接收到一个 WM_MOUSEMOVE 消息时，它把 wParam 和 MK_LBUTTON 进行按位与（ AND ）运算，来确定是否按下了左键。 wParam 也可以用于确定换档键的状态。例如，如果处理必须依赖于 Shift 和 Ctrl 键的状态，那么你可以使用如下所示的方法：

```
if (MK_SHIFT & wParam)
     if (MK_CONTROL & wParam)
          {
          [Shift and Ctrl keys are down]
          }
     else
          {
          [Shift key is down]
          }
else if (MK_CONTROL & wParam)
```

```
            {
            [Ctrl key is down]
            }
        else
            {
            [neither Shift nor Ctrl key is down]
            }
```

如果你想在程序中同时使用左右键，同时如果你还希望只有单键鼠标的用户也能使用你的程序，那么你可以这样来编写代码：Shift 与左键的组合使用等效于右键。在这种情况下，对鼠标键的处理可以采用如下所示的方法：

```
case WM_LBUTTONDOWN :
     if (!MK_SHIFT & wParam)
          {
          [left button logic]

          return 0 ;
          }
                    // fall through
case WM_RBUTTONDOWN :
     [right button logic]

     return 0 ;
```

Windows 函数 GetKeyState（在第 4 章中介绍过）可以使用虚拟键码 VK_LBUTTON、VK_RBUTTON、VK_MBUTTON、VK_SHIFT 和 VK_CNOTROL 来返回鼠标键与 Shift 键的状态。如果 GetKeyState 返回负值，则说明按下了鼠标键或者换档键。因为 GetKeyState 返回当前正在处理的鼠标键或者换档键的状态，所以所有状态信息同相应的消息是同步的。但是，正如不能把 GetKeyState 用于尚未按下的键一样，你也不能为尚未按下的鼠标键调用 GetKeyState 。不要这样做：

```
while (GetKeyState (VK_LBUTTON)>0) ;    //错误！！！
```

只有在你调用 GetKeyState 期间处理消息时，如果左键已经按下，才会报告键已经按下的消息。

6.2.3 双击鼠标键

双击鼠标键是指在短时间内按下并释放键两次，仅当双击发生在称为“双击时间”这一特定间隔内时才可以称为双击鼠标键。如果希望你的窗口过程能够收到双击键的鼠标消息，那么在调用 RegisterClass 初始化窗口类结构时，必须在窗口风格中包含 CS_DBLCLKS 标识符：

```
wndclass.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS ;
```

如果在窗口风格中未包含 CS_DBLCLKS ，而用户在短时间内双击了鼠标键，那么窗口过程接收到下面这些消息： WM_LBUTTONDOWN 、 WM_LBUTTONUP 、 WM_LBUTTONDOWN 和 WM_LBUTTONUP （窗口过程可能在这些键消息之前还收到了其它消息）。如果你想实现自己的双击逻辑，那么你可以使用 Windows 函数 GetMessageTime 获取 WM_LBUTTONDOWN 消息之间的相对时间。第 5 章将更详细地讨论这个函数。

如果你在窗口类中包含了 CS_DBLCLKS ，那么双击时窗口过程收到如下消息： WM_LBUTTONDOWN 、 WM_LBUTTONUP 、 WM_LBUTTONDBLCLK 和 WM_LBUTTONUP 。与前面的差别只在于一条 WM_LBUTTONDBLCLK 消息替换了第二个 WM_LBUTTONDOWN 消息。

如果双击中的第一次键操作完成单击的功能，那么双击这一消息是很容易处理的。第二次按键（ WM_LBUTTONDBLCLK 消息）则完成第一次按键以外的事情。例如，看看 Windows 资源管理器中是如何用鼠标来操作文件列表的。按一次键将选中文件， Windows 资源管理器用反转视频显示条加亮文件。双击则实现两个功能：第一次如单击那样选中文件；第二次则指示 Windows 资源管理器运行该文件，执行逻辑相当简单。如果双击中的第一次不执行单键功能，那么鼠标处理逻辑会变得非常复杂。

6.3 非客户区鼠标消息

在窗口的客户区内移动或按下鼠标键时，将产生 10 种消息。如果鼠标在窗口的客户区之外但还在窗口内， Windows 就给窗口过程发送一条“非客户区”鼠标消息。非客户区包括标题条、菜单和窗口滚动条。

通常，你不需要处理非客户区鼠标消息，而将这些消息传给 DefWindowProc ，从而使 Windows 执行系统功能。就这方面来说，非客户区鼠标消息类似于系统键盘消息 WM_SYSKEYDOWN 、 WM_SYSKEYUP 和 WM_SYSCHAR 。

非客户区鼠标消息几乎完全与客户区鼠标消息平行出现。消息中含有字母“ NC ”以表示是非客户区消息。如果鼠标在窗口的非客户区中移动，那么窗口过程接收到 WM_NCMOUSEMOVE 消息。鼠标键产生以下消息：

键	按下	释放	双击键
左	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
中	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
右	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

对非客户区鼠标消息， wParam 和 lParam 参数与客户区鼠标消息的 wParam 和 lParam 参数不同。 wParam 参数指明移动或者按鼠标键的非客户区。它设置为 WINDOWS.H 中定义的以 HT 开头的标识符之一（ HT 表示“命中测试”）。

lParam 参数的低位字为 x 坐标，高位字为 y 坐标，但是，它们是屏幕坐标，而不是像客户区鼠标消息那样指的是客户区坐标。对屏幕坐标，显示器左上角的 x 和 y 的值为 0 。当往右移时 x 的值增加，往下移时 y 的值增加（见图 6-3 ）。

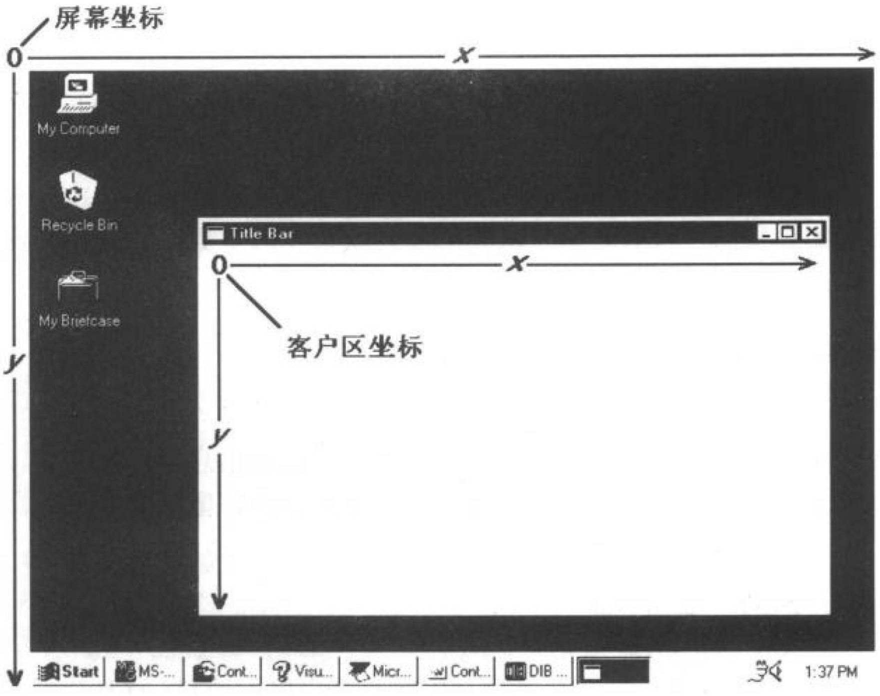


图 6-3 屏幕坐标与客户区坐标

你可以用两个 Windows 函数将屏幕坐标转换为客户区坐标或者反之：

```
ScreenToClient (hwnd, pPoint) ;
ClientToScreen (hwnd, pPoint) ;
```

pPoint 参数是指向 POINT 类型结构的远指针，这两个函数把转换得到的值存放在结构中而不保存原来的值。如果屏幕坐标点在窗口客户区上方，那么转换后的 y 坐标将为负值。类似地，如果屏幕坐标点在客户区的左方，那么用客户区坐标表示时 x 将为负值。

6.3.1 命中测试

如果你数一下，就可以知道我们已经介绍了 21 个鼠标消息中的 20 个，最后一个消息是 WM_NCHITTEST，它代表“非客户区命中测试”。此消息优先于所有其它的客户区和非客户区鼠标消息。lParam 参数含有鼠标位置的 x 和 y 屏幕坐标，wParam 参数没有用。

Windows 应用程序通常把这个消息传送给 DefWindowProc，然后 Windows 用 WM_NCHITTEST 消息产生基于鼠标位置的所有其它鼠标消息。对于非客户区鼠标消息，当处理 WM_NCHITTEST 时，从 DefWindowProc 返回的值成为鼠标消息中的 wParam 参数，这个值可以是任意非客户区鼠标消息的 wParam 值再加上以下内容：

HTCLIENT	客户区
HTNOWHERE	不在窗口中

HTTRANSPARENT	窗口由另一个窗口覆盖
HTERROR	使 DefWindowProc 产生蜂鸣声

如果 DefWindowProc 在其处理 WM_NCHITTEST 消息后返回 HTCLIENT，那么 Windows 把屏幕坐标转换为客户区坐标并产生客户区鼠标消息。

如果你还记得我们如何通过捕获 WM_SYSKEYDOWN 消息来禁止所有的系统键盘功能，那么你可能想我们可否通过捕获鼠标消息完成类似的事情。如果你在窗口过程中包含以下几条语句：

```
case WM_NCHITTEST :
    return (long)HTNOWHERE ;
```

就可以有效地禁止你窗口中的所有客户区和非客户区鼠标消息。这样一来，当鼠标在你的窗口（包括系统菜单图标、缩放按钮以及关闭按钮）中时，鼠标键将失效。

6.3.2 从消息产生消息

Windows 用 WM_NCHITTEST 消息产生所有其它鼠标消息，这种由消息引出其它消息的思想在 Windows 中是很普遍的。让我们来举个例子。你知道，如果你在一个 Windows 程序的系统菜单图标上双击一下，那么程序将会终止。双击产生一系列的 WM_NCHITTEST 消息。由于鼠标定位在系统菜单图标上，因此 DefWindowProc 将返回 HTSYSMENU 的值，并且 Windows 把 wParam 等于 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 消息放在消息队列中。

窗口过程通常把鼠标消息传递给 DefWidnowProc，当 DefWindowProc 接收到 wParam 参数等于 HTSYSMENU 的 WM_NCLBUTTONDBLCLK 消息时，它就把 wParam 参数等于 SC_CLOSE 的 WM_SYSCOMMAND 消息放入消息队列中（这个 WM_SYSCOMMAND 消息是在用户从系统菜单中选择 Close 时产生的）。同样，窗口过程也把这个消息传给 DefWindowProc，DefWindowProc 通过给窗口过程发送 WM_CLOSE 消息来处理该消息。

如果一个程序在终止之前要求来自用户的确认，那么窗口过程就需要捕获 WM_CLOSE，否则，DefWindowProc 调用 DestroyWindow 函数来处理 WM_CLOSE。除了其它处理，DestroyWindow 还给窗口过程发一条 WM_DESTROY 消息。窗口过程通常用下列代码来处理 WM_DESTROY 消息：

```
case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
```

PostQuitMessage 使 Windows 把 WM_QUIT 消息放入消息队列中，此消息永远不会到达窗口过程，因为它使 GetMessage 返回 0，终止消息循环，从而也终止了程序。

6.4 程序中的命中测试

我在前面讨论了 Windows 资源管理器如何响应鼠标的单击和双击。显然，程序必须确定用户用鼠标所指向的是哪一个文件，这叫做“命中测试”。正如 DefWindowProc 在处理 WM_NCHITTEST 消息时做一些命中测试一样，窗口过程经常必须在客户区中进行一些命中测试。一般来讲，命中测试中会使用 x 和 y 坐标值，它们由传到窗口过程的鼠标消息的 lParam 参数给出。

6.4.1 一个假想的例子

有这样一个例子，你的程序显示几列按字母顺序排列的文件列表。文件列表开始于客户区的顶端，客户区为 cxClient 个象素宽， cyClient 个象素高，每个字符高度为 cyChar 个象素。文件名保存在一个称为 szFileNames 的指向已排序的字符串指针数组中。

假定一列为 cxColWidth 个象素宽，那么每栏的文件数为：

```
iNumInCol = cyClient / cyChar ;
```

你接收到一个坐标为 cxMouse 和 cyMouse 的鼠标键消息， cxMouse 和 cyMouse 是从 lParam 中得来的。使用公式：

```
iColumn = cxMouse / cxColWidth ;
```

就可以确定鼠标指向哪一个文件名。相对于列顶端的文件名位置为：

```
iFromTop = cyMouse / cyChar ;
```

现在你就可以计算 szFileNames 数组的下标：

```
iIndex = iColumn * iNumInCol + iFromTop ;
```

很明显，如果 iIndex 超过了数组中的文件数，则表示用户是在显示器的空白区域内按鼠标键。

在许多情况下，命中测试要比本例更加复杂。例如，在一个使用可变字体尺寸的字处理程序（诸如 WORDPAD ）中，命中测试是非常复杂的。当在客户区显示一些东西时，你必须决定要显示的每个东西的坐标。在命中测试计算中，你必须从坐标找到对象。但是如果你要显示的对象是字符串，那么将涉及查找字符串中特定字符的位置。

6.4.2 示例程序

图 6-4 中所示的 CHECKER1 程序演示了一些简单的命中测试，此程序把客户区分为 5 × 5 的 25 个矩形。如果你在某个矩形中按下鼠标键，那么在该矩形中将出现一个 X 。如果你再按一次，那么 X 将被删除。

CHECKER1.MAK

```
#-----------------------
# CHECKER1.MAK make file
#-----------------------

checker1.exe : checker1.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker1.exe checker1.obj $(GUILIBS)

checker1.obj : checker1.c
     $(CC) $(CFLAGS) checker1.c
```

CHECKER1.C

```
/*-------------------------------------------------
   CHECKER1.C -- Mouse Hit-Test Demo Program No. 1
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx (hdc, x, y, NULL)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR  szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker1" ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL), IDI_APPLICATION ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Checker1 Mouse Hit-Test Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
```

```
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fState[DIVISIONS][DIVISIONS] ;
     static int  cxBlock, cyBlock ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;
     int         x, y ;

     switch (iMsg)
          {
          case WM_SIZE :
               cxBlock = LOWORD (lParam) / DIVISIONS ;
               cyBlock = HIWORD (lParam) / DIVISIONS ;
               return 0 ;

          case WM_LBUTTONDOWN :
               x = LOWORD (lParam) / cxBlock ;
               y = HIWORD (lParam) / cyBlock ;

               if (x < DIVISIONS && y < DIVISIONS)
                    {
                    fState [x][y] ^= 1 ;

                    rect.left   = x * cxBlock ;
                    rect.top    = y * cyBlock ;
                    rect.right  = (x + 1) * cxBlock ;
                    rect.bottom = (y + 1) * cyBlock ;

                    InvalidateRect (hwnd, &rect, FALSE) ;
                    }
               else
                    MessageBeep (0) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (x = 0 ; x < DIVISIONS ; x++)
                    for (y = 0 ; y < DIVISIONS ; y++)
                         {
                         Rectangle (hdc, x * cxBlock, y * cyBlock,
                                   (x + 1) * cxBlock, (y + 1) * cyBlock) ;

                         if (fState [x][y])
```

```
                              {
                              MoveTo (hdc,  x     * cxBlock,
                                            y     * cyBlock) ;
                              LineTo (hdc,  (x+1) * cxBlock,
                                            (y+1) * cyBlock) ;
                              MoveTo (hdc,  x     * cxBlock,
                                            (y+1) * cyBlock) ;
                              LineTo (hdc,  (x+1) * cxBlock,
                                            y     * cyBlock) ;
                              }
                         }
               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 6-4 CHECKER1程序

图 6-5 是 CHKECKER1 的显示，所有 25 个矩形的宽度和高度均相同。高度和宽度值保存在 cxBlock 和 cyBlock 中，当客户区大小发生改变时将重新对这些值进行计算。WM_LBUTTONDOWN 处理过程使用鼠标坐标来确定在哪个矩形中按下了键，它在 fState 数组中标志当前矩形的状态，并使该矩形区域失效，从而产生 WM_PAINT 消息。如果客户区的宽度和高度不能被 5 整除，那么在客户区的左边和下边将有一小条不能被矩形所覆盖。对出错情况，CHECKER1 通过调用 MessageBeep 响应此区域中的鼠标键操作。

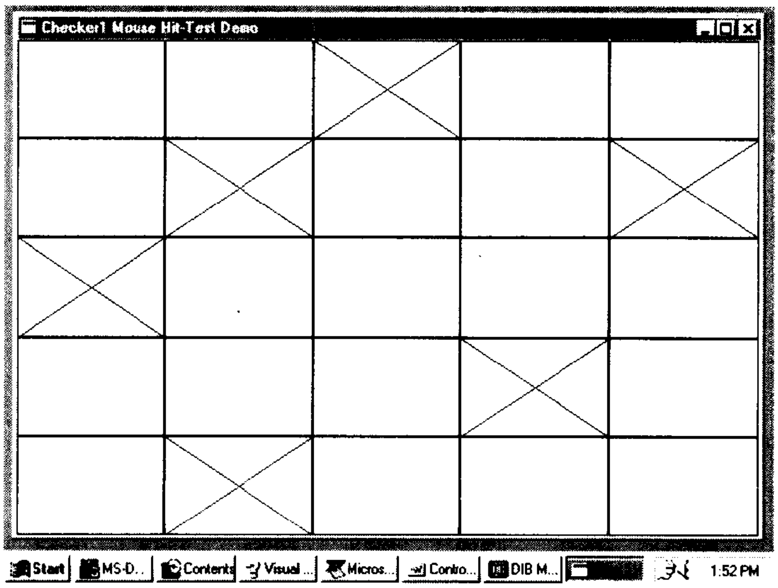


图 6-5 CHECKER1的显示

当 CHECKER1 收到 WM_PAINT 消息时，它通过 GDI 的 Rectangle 函数来重新绘制客户区。如果 fState 值被置为 1，那么 CHECKER1 将使用 MoveTo 和 LineTo 函数来绘制两条直线。在处理 WM_PAINT 期间，CHECKER1 在重新绘制之前并不检查每个矩形区域的有效性，尽管它可以这样做。检查有效性的一种方法是在循环中为每个矩形块构造 RECT 结构（使用 WM_LBUTTONDOWN 逻辑中相同的公式），并使用 IntersectRect 函数检查它是否同无效矩形（ps.rcPaint）相交。另一种方法是使用 PtInRect 来确定矩形的四个角是否在无效矩形内。

6.4.3 用键盘模拟鼠标

CHECKER1 只能在装有鼠标时才可运行。下面我们在程序中加入键盘接口，就如同第 5 章中对 SYSMETS 程序所做的那样。不过，在一个使用鼠标光标的程序中加入键盘接口要求我们必须处理鼠标光标的移动和显示问题。

即使没有安装鼠标，Windows 仍然可以显示一个鼠标光标。Windows 为这个光标保存了一个“显示计数”。如果安装了鼠标，显示计数被初始化为 0；否则，显示计数被初始化为－1。只有在显示计数为 0 或者正数时才显示鼠标光标，你可以通过调用

```
ShowCursor (TRUE) ;
```

来增加显示计数。也可以通过调用：

```
ShowCursor (FALSE) ;
```

来减少显示计数。

你在使用 ShowCursor 之前不需要确定是否安装了鼠标。如果你想显示鼠标光标而不管鼠标存在与否，那么只需要增加显示计数。在增加一次显示计数之后，再减少它时，如果未安装鼠标，则鼠标光标不再被显示，否则将仍然被显示。显示计数适用于整个 Windows，所以应该保证递增和递减相同的次数。

你可能想在你的窗口过程中使用下列的简单逻辑：

```
case WM_SETFOCUS:
    ShowCursor (TRUE) ;
    return 0 ;

case WM_KILLFOCUS:
    ShowCursor (FALSE) ;
    return 0 ;
```

窗口过程在窗口获得输入焦点时收到 WM_SETFOCUS 消息，在失去输入焦点时收到 WM_KILLFOCUS 消息。显示和隐藏鼠标光标有一个理想的次数。首先 WM_SETFOCUS 和 WM_KILLFOCUS 调用是平衡的——也就是说，窗口过程递增和递减相同次数的鼠标光标显示计数。其次，对没有安装鼠标的 Windows 系统，使用 WM_SETFOCUS 和 WM_KILLFOCUS 消息使得只有在窗口拥有输入焦点时光标才会可见，也只有在这时候用

户才可以使用你所设计的键盘接口来移动光标。

即使未安装鼠标， Windows 也会维护一个当前鼠标光标位置。如果没有安装鼠标时显示鼠标光标，那么它可能会出现在显示器的任何一个地方，在你移动光标之前，它将一直停留在那个地方。你可以通过使用：

```
GetCursorPos (pPoint) ;
```

来获取光标的位置，其中 pPoints 是一个指向 POINT 结构的远指针，函数使用鼠标的 x 和 y 坐标来填充 POINT 域。你可以通过使用：

```
SetCursorPos (x,y) ;
```

来设置光标位置。在这两种情况下， x 和 y 都是屏幕坐标，而不是客户区坐标（这是显然的，因为这些函数没有要求 hwnd 参数）。前面已经提到过，使用 ScreenToClient 和 ClientToScreen 能做到屏幕坐标与客户坐标的相互转换。

如果你在处理鼠标消息并转换客户区坐标时调用 GetCursorPos ，那么所得到的坐标与鼠标消息的 lParam 中的坐标稍微有些不同，鼠标消息的 lParam 中的坐标是消息产生时的鼠标坐标。

你或许想编写一个键盘逻辑：使用键盘方向键来移动鼠标光标，使用 Spacebar 和 Enter 键来模拟鼠标键。你肯定不希望每次击键只是将鼠标光标移动一个象素，这样做，当把鼠标光标从显示器的一边移动到另一边时，会使用户按住方向键的时间超过一分钟。

如果你需要实现鼠标光标的键盘接口并保持光标的精确定位能力，那么你可以采用下面的方式来处理击键消息：当按下方向键时，鼠标光标开始移动较慢，但会慢慢地加快。你也许还记得 WM_KEYDOWN 消息中的 lParam 参数标志着击键消息是否是重复活动的结果，这就是此参数的一个重要应用。

6.4.4 在 CHECKER 中加入键盘接口

图 6-6 所示的 CHECKER2 程序，除了包括键盘接口外，同 CHECKER1 是一样的。你可以使用左、右、上和下方向键在 25 个矩形之间移动光标， Home 键把光标移动到矩形的左上角， End 键把光标移动到矩形的右下角， Spacebar 和 Enter 键都能切换 X 标记。

CHECKER2.MAK

```
#------------------------
# CHECKER2.MAK make file
#------------------------

checker2.exe : checker2.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker2.exe checker2.obj $(GUILIBS)

checker2.obj : checker2.c
     $(CC) $(CFLAGS) checker2.c
```

CHECKER2.C

```
/*----------------------------------------------
   CHECKER2.C -- Mouse Hit-Test Demo Program No. 2
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx (hdc, x, y, NULL)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Checker2 Mouse Hit-Test Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fState[DIVISIONS][DIVISIONS] ;
     static int  cxBlock, cyBlock ;
```

```
HDC          hdc ;
PAINTSTRUCT ps ;
POINT        point ;
RECT         rect ;
int          x, y ;

switch (iMsg)
     {
     case WM_SIZE :
          cxBlock = LOWORD (lParam) / DIVISIONS ;
          cyBlock = HIWORD (lParam) / DIVISIONS ;
          return 0 ;

     case WM_SETFOCUS :
          ShowCursor (TRUE) ;
          return 0 ;

     case WM_KILLFOCUS :
          ShowCursor (FALSE) ;
          return 0 ;

     case WM_KEYDOWN :
          GetCursorPos (&point) ;
          ScreenToClient (hwnd, &point) ;

          x = max (0, min (DIVISIONS - 1, point.x / cxBlock)) ;
          y = max (0, min (DIVISIONS - 1, point.y / cyBlock)) ;

          switch (wParam)
               {
               case VK_UP :
                    y-- ;
                    break ;

               case VK_DOWN :
                    y++ ;
                    break ;

               case VK_LEFT :
                    x-- ;
                    break ;

               case VK_RIGHT :
                    x++ ;
                    break ;

               case VK_HOME :
                    x = y = 0 ;
                    break ;

               case VK_END :
                    x = y = DIVISIONS - 1 ;
                    break ;

               case VK_RETURN :
```

```
                    case VK_SPACE :
                         SendMessage (hwnd, WM_LBUTTONDOWN, MK_LBUTTON,
                                   MAKELONG (x * cxBlock, y * cyBlock)) ;
                         break ;
                    }
               x = (x + DIVISIONS) % DIVISIONS ;
               y = (y + DIVISIONS) % DIVISIONS ;

               point.x = x * cxBlock + cxBlock / 2 ;
               point.y = y * cyBlock + cyBlock / 2 ;

               ClientToScreen (hwnd, &point) ;
               SetCursorPos (point.x, point.y) ;
               return 0 ;

          case WM_LBUTTONDOWN :
               x = LOWORD (lParam) / cxBlock ;
               y = HIWORD (lParam) / cyBlock ;

               if (x < DIVISIONS && y < DIVISIONS)
                    {
                    fState[x][y] ^= 1 ;

                    rect.left   = x * cxBlock ;
                    rect.top    = y * cyBlock ;
                    rect.right  = (x + 1) * cxBlock ;
                    rect.bottom = (y + 1) * cyBlock ;

                    InvalidateRect (hwnd, &rect, FALSE) ;
                    }
               else
                    MessageBeep (0) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (x = 0 ; x < DIVISIONS ; x++)
                    for (y = 0 ; y < DIVISIONS ; y++)
                         {
                         Rectangle (hdc, x * cxBlock, y * cyBlock,
                                   (x + 1) * cxBlock, (y + 1) * cyBlock) ;

                         if (fState [x][y])
                              {
                              MoveTo (hdc, x     * cxBlock,
                                           y     * cyBlock) ;
                              LineTo (hdc, (x+1) * cxBlock,
                                           (y+1) * cyBlock) ;
                              MoveTo (hdc, x     * cxBlock,
                                           (y+1) * cyBlock) ;
                              LineTo (hdc, (x+1) * cxBlock,
                                           y     * cyBlock) ;
                              }
                         }
```

```
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 6-6 CHECKER2程序

CHECKER2 中的 WM_KEYDOWN 逻辑决定光标的位置（GetCursorPos），把屏幕坐标转换为客户区坐标（ScreenToClient）并用矩形块的宽度和高度来除这个坐标。这会产生指示矩形位置的 x 和 y 值（5 × 5 数组）。当按下一个键时，鼠标光标可能在，也可能不在客户区中，从而 x 和 y 必须经 min 和 max 宏处理以保证它们的范围是 0 到 4 。

对方向键， CHECKER2 近似地增加或减少 x 和 y 。如果是 Enter 键（VK_PRETUR）或 Spacebar（VK_SPACE），那么 CHECKER2 使用 SendMessage 把 WM_LBUTTONDOWN 消息发送给它自身。这种技术类似于在第 5 章 SYSMETS 程序中把键盘接口加到窗口滚动条时所使用的方法。 WM_KEYDOWN 逻辑是通过计算指向矩形中心的客户区坐标，再转换成屏幕坐标（ClientToScreen），然后设置光标位置（SetCursorPos）来实现的。

6.4.5 将子窗口用于命中测试

有些程序，像 Windows PAINT 程序，把客户区划分为几个更小的逻辑区域。图 6-7 中所示的是 PAINT 程序，在其左边有一个基于图标的菜单，其底部有一个颜色菜单，当在这两个菜单上进行命中测试时， PAINT 必须在确定用户选中的菜单项之前记住菜单的位置。

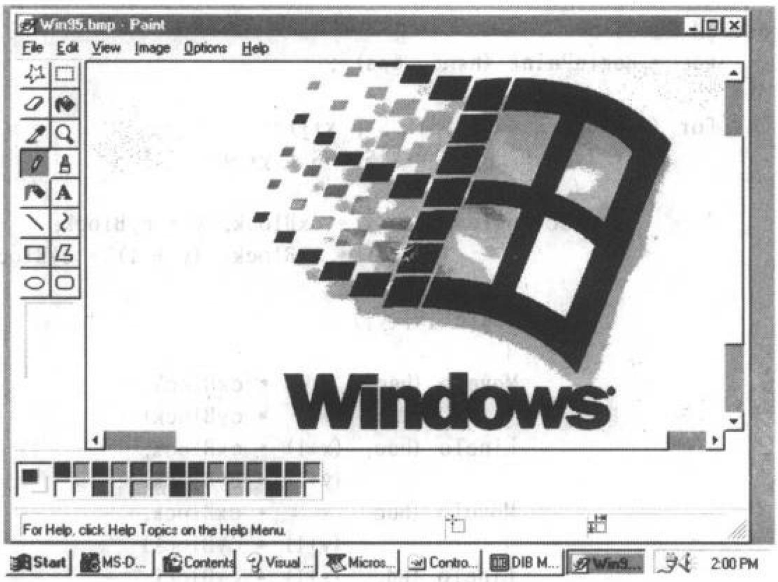


图 6-7 Windows PAINT程序

不过，也可能不需要这么做。实际上，PAINT 通过使用子窗口简化了菜单的绘制和命中测试。子窗口把整个矩形区域划分为几个更小的矩形区，每个子窗口有自己的窗口句柄，窗口过程和客户区，每个窗口过程接收只适用于它的子窗口的鼠标消息。鼠标消息中的 lParam 参数含有相当于该子窗口客户区左上角的坐标，而不是其父窗口客户区左上角的坐标。

这种方法使用的子窗口有助于程序的结构化和模块化。如果子窗口使用不同的窗口类，那么每个窗口有它自己的窗口过程。不同的窗口也可以定义不同的背景颜色和不同的缺省光标。在第 8 章中，我们将看到"子窗口控制"——滚动条、按钮和编辑框等预定义的子窗口。现在，我们说明在 CHECKER 程序中是如何使用子窗口的。

6.4.6 CHECKER 中的子窗口

图 6-8 中所示的 CHECKER3 程序，这一版本创建了 25 个处理鼠标消息的子窗口。它没有键盘接口，但是很容易加进去。

CHECKER3 有两个窗口过程，分别称为 WndProc 和 ChildWndProc 。 WndProc 仍是主（父）窗口的窗口过程，ChildWndProc 是 25 个子窗口的窗口过程，这两个窗口过程都必须定义为 CALLBACK 函数。

因为窗口过程是由 RegisterClass 在 Windows 中登录的窗口类结构所定义的，所以 CHECKER3.C 中的两个窗口过程要求两个窗口类。前一个窗口类用于主窗口，名字为" Checker3 "，第二个窗口类叫做" Checker3_Child "。

当在 WinMain 中登录" Checker3_Child "时，可以简单地重用 wndclass 结构变量中的大多数域。lpszClassName 域被置为" Checker3_Child "（类名）。lpfnWndProc 域被置为 ChildWndProc ，即本窗口类的窗口过程。因为子窗口不使用图标，因此 hIcon 域被置为 NULL 。对" Checker3_Child "窗口类，wndclass 结构变量中的 cbWndExtra 域设置为 2 字节，或者更精确地说应该是 sizeof(WORD)个字节。这个域告诉 Windows ，在它为每个基于此窗口类的窗口维护的结构中保留 2 字节的额外空间，此空间可以用于存放每个子窗口各自的不同信息。

CHECKER3.MAK

```
#-----------------------
# CHECKER3.MAK make file
#-----------------------

checker3.exe : checker3.obj
     $(LINKER) $(GUIFLAGS) -OUT:checker3.exe checker3.obj $(GUILIBS)

checker3.obj : checker3.c
     $(CC) $(CFLAGS) checker3.c
```

CHECKER3.C

```
/*-------------------------------------------------
   CHECKER3.C -- Mouse Hit-Test Demo Program No. 3
```

```
                 (c) Charles Petzold, 1996
  -----------------------------------------------*/

#include <windows.h>

#define DIVISIONS 5
#define MoveTo(hdc, x, y) MoveToEx (hdc, x, y, NULL)

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ChildWndProc (HWND, UINT, WPARAM, LPARAM) ;

char    szChildClass[] = "Checker3_Child" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Checker3" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     wndclass.lpfnWndProc   = ChildWndProc ;
     wndclass.cbWndExtra    = sizeof (WORD) ;
     wndclass.hIcon         = NULL ;
     wndclass.lpszClassName = szChildClass ;
     wndclass.hIconSm       = NULL ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Checker3 Mouse Hit-Test Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
```

```
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndChild[DIVISIONS][DIVISIONS] ;
     int         cxBlock, cyBlock, x, y ;

     switch (iMsg)
          {
          case WM_CREATE :
               for (x = 0 ; x < DIVISIONS ; x++)
                    for (y = 0 ; y < DIVISIONS ; y++)
                         {
                         hwndChild[x][y] = CreateWindow (szChildClass, NULL,
                              WS_CHILDWINDOW | WS_VISIBLE,
                              0, 0, 0, 0,
                              hwnd, (HMENU) (y << 8 | x),
                              (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE),
                              NULL) ;
                         }
               return 0 ;

          case WM_SIZE :
               cxBlock = LOWORD (lParam) / DIVISIONS ;
               cyBlock = HIWORD (lParam) / DIVISIONS ;

               for (x = 0 ; x < DIVISIONS ; x++)
                    for (y = 0 ; y < DIVISIONS ; y++)
                         MoveWindow (hwndChild[x][y],
                              x * cxBlock, y * cyBlock,
                              cxBlock, cyBlock, TRUE) ;
               return 0 ;

          case WM_LBUTTONDOWN :
               MessageBeep (0) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

LRESULT CALLBACK ChildWndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_CREATE :
```

```
            SetWindowWord (hwnd, 0, 0) ;        // on/off flag
            return 0 ;

        case WM_LBUTTONDOWN :
            SetWindowWord (hwnd, 0, 1 ^ GetWindowWord (hwnd, 0)) ;
            InvalidateRect (hwnd, NULL, FALSE) ;
            return 0 ;

        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            GetClientRect (hwnd, &rect) ;
            Rectangle (hdc, 0, 0, rect.right, rect.bottom) ;

            if (GetWindowWord (hwnd, 0))
                 {
                 MoveTo (hdc, 0,          0) ;
                 LineTo (hdc, rect.right, rect.bottom) ;
                 MoveTo (hdc, 0,          rect.bottom) ;
                 LineTo (hdc, rect.right, 0) ;
                 }

            EndPaint (hwnd, &ps) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 6-8 CHECKER3程序

在 WinMain 中，CreateWindow 调用创建基于“Checker3”类的主窗口。当 WndProc 收到WM_CREATE消息时，它调用CreateWindow 25次以创建25个基于“Checker3_Child”类的子窗口。下表是创建主窗口的 WinMain 中的 CreateWindow 调用的参数和创建 25 个子窗口的 WndProc 中的 CreateWindow 调用的参数间的比较。

参数	主窗口	子窗口
窗口类	“Checker3”	“Checker3_Child”
窗口标题	“Checker3...”	NULL
窗口风格	WS_OVERLAPPEDWINDOW	WS_CHILDWINDOW | WS_VISIBLE
水平位置	CW_USEDEFAULT	0
垂直位置	CW_USEDEFAULT	0
宽度	CW_USEDEFAULT	0
高度	CW_USEDEFAULT	0
父窗口句柄	NULL	hwnd
菜单句柄/子 ID	NULL	y << 8 | x
实例句柄	hInstance	GetWindowWord (hwnd, GWL_HINSTANCE)
额外参数	NULL	NULL

通常情况下，子窗口要求有位置、宽度和高度参数，但是在 CHECKER3 中的子窗口是由 WndProc 确定位置和大小的。对主窗口，它的父窗口为 NULL 。当使用 CreateWindow 调用创建子窗口时，要求父窗口句柄。

主窗口没有菜单，所以该参数为 NULL ；对子窗口，同一位置的参数称为子 ID （子窗口标识），它是唯一能标识该窗口的数值。在使用子窗口控制时子 ID 显得更为重要，因为发送到父窗口的消息由这个子 ID 来标识；对 CHECKER3 来说，子 ID 用于标识每个子窗口在 5 × 5 的主窗口中的位置。

在这两种情况下，实例句柄都为 hInstance 。当创建子窗口时，使用 GetWindowWord 函数从 Windows 为该窗口保留的结构中取出 hInstance 值(除了使用 GetWindowWord 之外，我还把 hInstance 的值保存在一个全局变量中，然后直接使用它）。

每个子窗口都有一个不同的保存在 hWndChild 数组中的窗口句柄。当 WndProc 收到 WM_SIZE 消息时，它为 25 个子窗口中的每一个调用 MoveWindow，其参数指明相对于父窗口客户区坐标的子窗口的左上角坐标、子窗口的宽度与高度以及子窗口是否需要重画的信息。

窗口过程 ChildWndProc 处理所有 25 个子窗口的消息，ChildWndProc 的 hWnd 参数是收到消息的子窗口句柄。当 ChildWndProc 处理 WM_CREATE 消息时（因为有 25 个子窗口，所以发生 25 次处理），它用 SetWindowsWord 将 0 填充到窗口结构中的额外域中。（记得我们在定义窗口类结构时使用 cbWndExtra 域保留了此空间）。ChildWndProc 使用这个值来保存矩形的状态（有 X 或者没有 X ）。当在一个子窗口内按下鼠标键时，WM_LBUTTONDOWN 逻辑修改这个值（从 0 变到 1 或者从 1 变到 0 ）使整个子窗口客户区无效，此区域是在其中按下键的那个矩形。WM_PAINT 的处理很普通，因为它所绘制的矩形同客户区一样大。

因为 CHECKER3 的 C 源代码文件和.EXE 文件比 CHECKER1 大，不能说 CHECKER3 比 CHECKER1 简单；但是在 CHECKER3 中不需要命中测试。如果 CHECKER3 中的子窗口得到 WM_LBUTTONDOWN 消息，即说明它被命中了。

如果你想在 CHECKER3 中加入键盘接口，那么主窗口会接收到键盘消息，因为它有输入焦点。在第 8 章中我们将进一步讨论子窗口。

6.5 鼠标的捕获

窗口过程通常仅当鼠标光标被定位在窗口的客户区或非客户区时才能收到鼠标消息。程序也许需要当鼠标在窗口外时也能接收鼠标消息，这就是所谓程序对鼠标的“捕获”。

6.5.1 画一个矩形

为了说明为什么捕获鼠标是必要的，让我们来看一看图 6-9 中所示的 BLOKOUT1 程序。

BLOKOUT1.MAK

```
#----------------------
# BLOKOUT1.MAK make file
#----------------------

blokout1.exe : blokout1.obj
     $(LINKER) $(GUIFLAGS) -OUT:blokout1.exe blokout1.obj $(GUILIBS)

blokout1.obj : blokout1.c
     $(CC) $(CFLAGS) blokout1.c
```

BLOKOUT1.C

```
/*-----------------------------------------
   BLOKOUT1.C -- Mouse Button Demo Program
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BlokOut1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW * CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Mouse Button Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
```

```
     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
     {
     HDC hdc ;

     hdc = GetDC (hwnd) ;

     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;

     ReleaseDC (hwnd, hdc) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;

     switch (iMsg)
          {
          case WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD (lParam) ;
               ptBeg.y = ptEnd.y = HIWORD (lParam) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

               fBlocking = TRUE ;
               return 0 ;

          case WM_MOUSEMOVE :
               if (fBlocking)
                    {
                    SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                    ptEnd.x = LOWORD (lParam) ;
                    ptEnd.y = HIWORD (lParam) ;

                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    }
               return 0 ;

          case WM_LBUTTONUP :
```

```
            if (fBlocking)
                 {
                 DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                 ptBoxBeg   = ptBeg ;
                 ptBoxEnd.x = LOWORD (lParam) ;
                 ptBoxEnd.y = HIWORD (lParam) ;

                 SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

                 fBlocking = FALSE ;
                 fValidBox  = TRUE ;

                 InvalidateRect (hwnd, NULL, TRUE) ;
                 }
            return 0 ;

       case WM_CHAR :
            if (fBlocking & wParam == '\x1B')       // ie, Escape
                 {
                 DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

                 SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

                 fBlocking = FALSE ;
                 }
            return 0 ;

       case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            if (fValidBox)
                 {
                 SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                 Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                                 ptBoxEnd.x, ptBoxEnd.y) ;
                 }

            if (fBlocking)
                 {
                 SetROP2 (hdc, R2_NOT) ;
                 SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                 Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
                 }

            EndPaint (hwnd, &ps) ;
            return 0 ;

       case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
       }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
  }
```

图 6-9 BLOKOUT1程序

这个程序演示了在 Windows 绘图程序中实现的一些功能。你从按下鼠标左键来指示矩

形的一个角开始，然后拖动鼠标，程序绘制出一个矩形轮廓，其另一个角位于当前鼠标的位置。当你释放鼠标时，程序将填充矩形。图6-10显示了一个已经画出的矩形和另一个正在画的矩形。

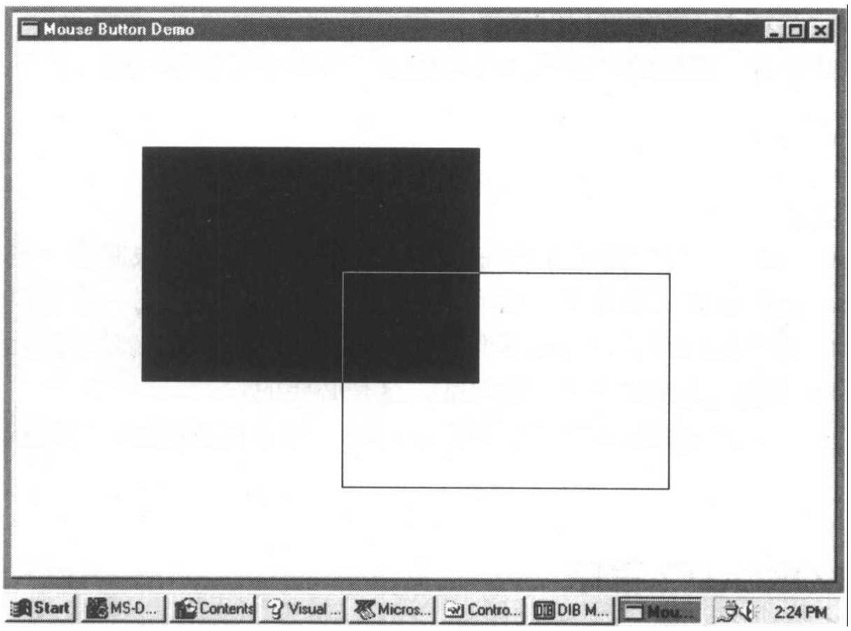


图 6-10 BLOKOUT1的显示

当按下左键时，BLOKOUT1保存鼠标的坐标并首次调用DrawBoxOutline。函数使用光栅操作R2_NOT绘制一个矩形，这一操作翻转客户区的颜色。在接下来的WM_MOUSEMOVE消息期间，程序再次绘制同样的矩形以清除前一个，然后它使用新的鼠标坐标绘制一个新的矩形。最后，当BLOKOUT1收到一个WM_LBUTTONUP消息时，保存鼠标坐标，使窗口无效，产生一个WM_PAINT消息以显示修改后的矩形。

那么，问题在哪儿呢？

试一试下面的方法：在BLOKOUT1的客户区内按下左鼠标键，然后将光标移动到窗口之外，程序停止接收WM_MOUSEMOVE消息。现在释放键，BLOKOUT1得不到WM_LBUTTONUP消息，因为此时光标位于客户区之外。将光标移动到BLOKOUT1的客户区内时，窗口过程仍然认为键处于按下状态。

这不太好，程序不知道发生了什么。

6.5.2 捕获的解决方案

BLOKOUT1包括了一些常见的程序功能，但是它的代码显然有缺陷，这就是要引入鼠标捕获来对付的一类问题。如果用户正在拖动鼠标，那么当鼠标短时间被拖出窗口时应该没有什么大问题，程序应该仍然控制着鼠标。

捕获鼠标要比放置一个老鼠夹子容易一些，你只要调用：

```
SetCapture (hwnd) ;
```

在这个函数调用之后，Windows 将所有鼠标消息发给窗口句柄为 hwnd 的窗口过程。鼠标消息总是客户区消息，即使当鼠标目前正位于窗口的一个非客户区。lParam 参数将指示鼠标在客户区坐标中的位置。不过，当鼠标位于客户区的左边或者上方时，这些 x 和 y 坐标可以是负的。

当鼠标被捕获时，系统键盘函数也将被禁止。当你想释放鼠标时，调用：

```
ReleaseCapture () ;
```

从而使处理恢复正常。

在 Windows 95 下，鼠标捕获要比在以前的 Windows 版本中限制多一些。特别地，如果鼠标被捕获，而鼠标键目前并未被按下，并且鼠标光标移到了另一个窗口上，那么将不是由捕获鼠标的那个窗口而是由光标下面的窗口来接收鼠标消息。对于防止一个程序在捕获鼠标之后不释放它而引起整个系统的混乱，这是必要的。

换句话说，只有当鼠标键在你的客户区中被按下时才捕获鼠标；当键被释放时，才释放鼠标捕获。

6.5.3 BLOKOUT2 程序

演示鼠标捕获的 BLOKOUT2 程序如图 6-11 所示。

BLOKOUT2.MAK

```
#-----------------------
# BLOKOUT2.MAK make file
#-----------------------

blokout2.exe : blokout2.obj
     $(LINKER) $(GUIFLAGS) -OUT:blokout2.exe blokout2.obj $(GUILIBS)

blokout2.obj : blokout2.c
     $(CC) $(CFLAGS) blokout2.c
```

BLOKOUT2.C

```
/*---------------------------------------------------
   BLOKOUT2.C -- Mouse Button & Capture Demo Program
                 (c) Charles Petzold, 1996
  ---------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BlokOut2" ;
```

```
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Mouse Button & Capture Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
     {
     HDC hdc ;

     hdc = GetDC (hwnd) ;

     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;

     ReleaseDC (hwnd, hdc) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
```

```
switch (iMsg)
     {
     case WM_LBUTTONDOWN :
          ptBeg.x = ptEnd.x = LOWORD (lParam) ;
          ptBeg.y = ptEnd.y = HIWORD (lParam) ;
          DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

          SetCapture (hwnd) ;
          SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

          fBlocking = TRUE ;
          return 0 ;

     case WM_MOUSEMOVE :
          if (fBlocking)
               {
               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               ptEnd.x = LOWORD (lParam) ;
               ptEnd.y = HIWORD (lParam) ;

               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
               }
          return 0 ;

     case WM_LBUTTONUP :
          if (fBlockin g)
               {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;

               ptBoxBeg   = ptBeg ;
               ptBoxEnd.x = LOWORD (lParam) ;
               ptBoxEnd.y = HIWORD (lParam) ;

               ReleaseCapture () ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               fBlocking = FALSE ;
               fValidBox  = TRUE ;

               InvalidateRect (hwnd, NULL, TRUE) ;
               }
          return 0 ;

     case WM_CHAR :
          if (fBlocking & wParam == '\x1B')       // i.e., Escape
               {
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
               ReleaseCapture () ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               fBlocking = FALSE ;
```

```
            }
            return 0 ;

        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            if (fValidBox)
                {
                SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                                ptBoxEnd.x, ptBoxEnd.y) ;
                }

            if (fBlocking)
                {
                SetROP2 (hdc, R2_NOT) ;
                SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
                }

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 6-11 BLOKOUT2程序

BLOKOUT2 程序和 BLOKOUT1 程序一样，只是多了三行代码：一个在 WM_LBUTTONDOWN 消息期间对 SetCapture 的调用和在 WM_LBUTTONUP 以及 WM_CHAR 消息期间对 ReleaseCapture 的调用（对 WM_CHAR 消息的处理允许用户按下 Escape 键时释放鼠标捕获）。

试试这么做：使窗口小于屏幕的大小，开始在客户区之内画出矩形的轮廓，然后将鼠标光标向右下方向移动到客户区外，最后释放鼠标键。程序将得到整个矩形的坐标，只需放大窗口就可以观察到。

捕获鼠标并非只适用于那些古怪的应用程序。如果你需要每当在客户区按下一个鼠标键时都能够跟踪 WM_MOUSEMOVE 消息，并直到鼠标键被释放为止，那么你就应该捕获鼠标。这样将简化你的程序，同时又符合用户的期望。

第7章 定 时 器

Windows 定时器是一种输入设备，周期性地在指定间隔的时间过去时通知应用程序。你的程序将时间间隔告诉 Windows，例如“每 10 秒钟通知我一声。”，然后 Windows 给你的程序发送周期性发生的 WM_TIMER 消息以表示时间间隔到了。

初看起来，Windows 定时器似乎不如键盘和鼠标设备重要，对许多应用程序来说当然如此。但是，定时器比你可能认为的要重要得多，它不只用于计时程序，比如出现在工具条中的 Windows 时钟（这一章包括两个时钟程序——一个是模拟时钟，另一个是数字时钟）。下面是 Windows 定时器的其它应用，有些可能并不那么明显：

- 多任务——虽然 Windows 95 是一个抢占式的多任务环境，有时程序尽快将控制返回给 Windows 会效率更高。如果一个程序必须进行大量的处理，那么它可以将作业分成小块，每接收到一条 WM_TIMER 消息处理一块（我将在第 14 章中对此做更多地讨论）。
- 维护更新过的状态报告——程序可以利用定时器来显示持续变化信息的“实时”更新，比如关于系统资源的变化或者某个任务的进展情况。
- 实现“自动保存”功能——定时器提示 Windows 程序在指定的时间过去后把用户的工作保存到磁盘上。
- 终止程序演示版本的运行——一些程序的演示版本在其开始后的，比如说，30 分钟结束。如果时间已到，那么定时器就会通知应用程序。
- 步进移动——游戏中的图形对象或计算机辅助教学程序中的连续显示需要按指定的速率来处理。利用定时器可以消除由于微处理器速度不同而造成的不一致。
- 多媒体——播放 CD 音频、声音或者音乐的程序通常在后台播放音频数据。一个程序可以使用定时器来周期性地检查已播放了多少音频数据并据此协调屏幕上的视觉信息。

本章还讨论了定时器之外的其它 Windows 编程问题。在讨论窗口过程时，我们已经遇到过回调函数的概念，在定时器编程中也有回调函数的使用。本章还讨论了程序不能访问定时器时产生的问题，这一点与 Windows 以前的版本已经不太一样了，但是这里的方法仍然可以用在其它程序的错误处理中。最后提供了一个示例程序，它涉及“弹出式”的窗口类型的使用、对 WIN.INI 文件的访问以获取关于国际时钟和数据格式的信息、利用三角函数来有效地旋转屏幕上的图形对象等问题，这些问题显然是与定时器没什么太大关系的。

7.1 定时器基础

你可以通过调用 SetTimer 函数为你的 Windows 程序分配一个定时器。SetTimer 有一个时间间隔范围为 1 毫秒到 4,294,967,295 毫秒(将近 50 天）的整型参数，这个值指示 Windows 把 WM_TIMER 消息发送给程序的速率。例如，如果间隔为 1000 毫秒，那么 Windows 将每秒给程序发送一条 WM_TIMER 消息。

当你的程序用完定时器时，它调用 KillTimer 函数停止定时器消息。在处理 WM_TIMER 消息时，你可以通过调用 KillTimer 函数来编写一个“瞬间”定时器。KillTimer 调用清除消息队列中尚未被处理的 WM_TIMER 消息，从而使程序在调用 KillTimer 之后就不会再接收到 WM_TIMER 消息。

7.1.1 系统和定时器

Windows定时器是PC的硬件和ROM BIOS构造的定时器逻辑的一种相对简单的扩展。PC 的 ROM BIOS 初始化定时器芯片来产生硬件中断，这个中断有时被称为“时钟滴答”或者“定时器滴答”中断。这些中断每 54.925 毫秒产生一次，或者大约每秒 18.2 次。一些为 MS-DOS 编写的程序自己捕获这个硬件中断以实现时钟和定时器。

基于 Windows 的程序不这么做。Windows 本身处理硬件中断，这样应用程序就不必进行处理。对于目前拥有定时器的每个程序，Windows 通过减少刚开始由 SetTimer 调用传入的计数来处理定时器中断。当这个计数减到 0 时，Windows 在适当的应用程序队列中放置一条 WM_TIMER 消息，并将计数重新置为其最初的值。

因为 Windows 应用程序从正常的消息队列中获取 WM_TIMER 消息，所以你的程序在进行其它处理时不必担心被偶然的 WM_TIMER 消息中断了程序。在这方面，定时器类似于键盘和鼠标。驱动程序处理异步硬件中断事件，Windows 把这些事件翻译为规整的、结构化和顺序化的消息。

Windows 定时器具有基于 PC 的定时器一样的 54.925 毫秒的分辨率，这一事实有两个重要的结果：

- 使用单个定时器时，Windows 应用程序不能接收速率高于每秒 18.2 次的 WM_TIMER 消息。
- 在 SetTimer 调用中指定的时间间隔总是截尾为时钟滴答的整数倍。例如，1000 毫秒的间隔除以 54.925 毫秒，得到 18.207 个时钟滴答，截尾后为 18 个时钟滴答，它实际上为 989 毫秒。对每个小于 55 毫秒的间隔，每个时钟滴答都产生一条 WM_TIMER 消息。

7.1.2 定时器消息不是异步的

我已经提到过，为 IBM PC 及其兼容机编写的非 Windows 程序能通过截取硬件中断来

使用定时器滴答中断。当硬件中断产生时，当前运行的程序被挂起，控制转到硬中断处理程序中。在中断处理程序完成之后，再把控制返回到被中断的程序。

同硬件键盘和鼠标中断一样，硬件定时器中断有时也被称为异步中断，因为对于它所中断的程序来说它是随机产生的（实际上，对定时器中断来说，“等时”这个词要比“异步”更加确切，因为中断是以相等的时间间隔发生的。但是对于其它处理来说，中断仍然是异步的）。

虽然 Windows 也处理异步定时器中断，但是 Windows 送到应用程序的 WM_TIMER 消息却不是异步的。WM_TIMER 消息放在正常的消息队列之中，和其它消息一起参加排序，因此，如果在 SetTimer 调用中指定间隔为 1000 毫秒，那么不能保证程序每 1000 毫秒或者 989 毫秒就会收到一条 WM_TIMER 消息。如果其它程序的运行时间超过一秒，在此期间，你的程序将收不到任何 WM_TIMER 消息。你可以使用本章的程序来演示这一点。事实上，Windows 对 WM_TIMER 消息的处理非常类似于对 WM_PAINT 消息的处理，这两个消息都是低优先级的，程序只有在消息队列中没有其它消息时才接收它们。

WM_TIMER 还在另一方面同 WM_PAINT 相似：Windows 不能持续向消息队列中放入多条 WM_TIMER 消息，而是将多条的 WM_TIMER 消息组合成一条消息。因此，应用程序不会一次收到多条这样的消息，尽管可能在短时间内得到两条 WM_TIMER 消息。应用程序不能确定这种处理所导致的 WM_TIMER 消息“丢失”的数目。

如果你按住一个时钟程序的标题条，并在屏幕上移动该窗口，那么时钟将停止接收 WM_TIMER 消息。当时钟重新得到控制权时，它将直接跳到正确的时间，但这并不是因为它一次得到了几条 WM_TIMER 消息。程序必须确定实际的时间并自己进行设置。WM_TIMER 消息仅仅在需要更新时才提示程序，程序本身不能通过统计 WM_TIMER 消息的数目来保持时间（在本章后面，我们将编写两个每秒更新一次的时钟程序，并可以看到如何作到这一点）。

为方便起见，下面在讨论时钟时，我将使用“每秒得到一次 WM_TIMER 消息”这样的词句，但是请记住，这些消息并非精确的时钟滴答中断。

7.2 定时器的使用：三种方法

如果你需要在整个程序期间使用定时器，那么你可以在 WinMain 函数中或者在处理 WM_CREATE 消息时调用 SetTimer，并在响应 WM_DESTROY 消息时调用 KillTimer。在 WinMain 中设置定时器的做法提供了没有可用定时器时的最简单的错误处理。根据调用 SetTimer 时使用的参数，可以用以下三种方法之一使用定时器。

7.2.1 方法一

这是最方便的一种方法，它让 Windows 把 WM_TIMER 消息发送到应用程序的窗口过程中，SetTimer 调用的格式如下：

```
SetTimer (hwnd, 1, wMsecInterval, NULL) ;
```

第一个参数是其窗口过程将接收 WM_TIMER 消息的窗口的句柄。第二个参数是定时器 ID，它是一个非 0 数，在整个例子中假定为 1 。第三个参数是一个 32 位无符号整数，以毫秒为单位指定一个时间间隔，一个 60000 的值将使 Windows 每分钟发送一次 WM_TIMER 消息。

你可以通过调用：

```
KillTimer (hwnd, 1) ;
```

在任何时刻停止 WM_TIMER 消息（即使正在处理 WM_TIMER 消息）。此函数的第二个参数是 SetTimer 调用中所用的同一个定时器。在终止程序之前，你应该响应 WM_DESTROY 消息停止任何活动的定时器。

当你的窗口过程收到一个 WM_TIMER 消息时， wParam 参数等于定时器的 ID 值（上述情形为 1 ）， lParam 参数为 0 。如果需要设置多个定时器，那么对每个定时器都使用不同的定时器 ID 。 wParam 的值将随传递到窗口过程的 WM_TIMER 消息的不同而不同。为了使程序具有更好的可读性，你可以使用#define 语句定义不同的定时器 ID ：

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

然后你可以使用两个 SetTimer 调用来设置两个定时器：

```
SetTimer (hwnd, TIMER_SEC, 1000, NULL) ;
SetTimer (hwnd, TIMER_MIN, 6000, NULL) ;
```

WM_TIMER 的处理如下所示：

```
case WM_TIMER:
    switch  (wParam)
        {
        case TIMER_SEC:
            [每秒一次的处理]
            break;
        case TIMER_MIN:
            [每分钟一次的处理]
            break;
        }
    return 0;
```

如果你要为已经存在的定时器设置不同的时间间隔，那么停止该定时器并再次调用 SetTimer 。假定定时器 ID 为 1 ，则代码如下：

```
KillTimer (hwnd, 1) ;
SetTimer (hwnd, 1, iMsecInterval, NULL) ;
```

iMsecInterval 参数是以毫秒为单位的新时间间隔。如果在一个时钟程序中有一种是否显示秒的选择，那么你也许会使用这一技术。你只需简单地在 1000 毫秒和 60,000 毫秒这两个

值中选择一个作为定时器的时间间隔。

7.2.1.1 没有可用定时器时的处理

在 Windows 的早期，整个系统中最多只有 16 个定时器处于活动状态。最早， Windows 的用户是通过运行尽可能多的 Clock 程序而发现这个问题的，当运行第 17 个时钟时，得到一个出错信息“没有更多的时钟或定时器！” Windows 3.0 将定时器的数目加倍到 32 个，而 Windows 95 完全没有了这种限制，但是在你的 Windows 程序中编写大量的错误处理代码仍然是一种好的习惯。为了清楚起见，在我的早期示例代码中并没有这么作，不过，这里也正是讨论如何处理一些基本问题的合适地方。

如果没有定时器可用， SetTimer 将返回 NULL 。你的程序也许在没有定时器的情况下也能正常运行，但是如果你需要定时器（如一个时钟程序当然会需要定时器一样），那么如果不能得到一个定时器，应用程序将别无选择而只好终止。如果你在 WinMain 中调用 SetTimer ，那么你可以简单地通过从 WinMain 中返回 FALSE 来终止程序。

让我们假定你需要一个 1000 毫秒的定时器。在调用 CreateWIndow 之后，在消息循环之前可能有这么一条语句：

```
if (!SetTimer (hwnd, 1,1000, NULL))
    return FALSE;
```

这种终止方法不够友好，用户退出后不知道为何不能装入应用程序。使用 Windows 消息框显示一条信息是一种更友好的方法，实现起来也相当容易（关于消息框的完整讨论请你等到第 12 章，不过从这里你就可以开始了）。

消息框是在显示器中间显示的弹出式窗口。消息框有一个标题条但没有缩放区，标题条通常含有应用程序的名称。消息框还含有一条信息和一、二或者三个按钮（分别为 OK 、 Retry 、 Cancel 、 Yes 、 No 等等的某种组合）。消息框还含有一个预定义图标：小写 i（它代表“ information ”）、一个感叹号、一个问号或者一个句号。在运行 Windows 时，可以看到许多消息框。

下面的一段代码在 SetTimer 分配一个定时器失败时产生一个消息框：

```
if (!SetTimer (hwnd, 1, 100, NULL))
    {
    MessageBox (hwnd,
        "Too many clocks or timers!",
        "Program Name",
        MB_ICONEXCLAMATION | MB_OK) ;
    return FALSE;
    }
```

此消息框如图 7-1 所示。当用户按下 Enter 键或者按下 OK 按钮时， WinMain 将由于返回 FALSE 而终止。

缺省地，消息框是“应用程序模态”窗口，这意味着用户必须在应用程序继续运行之前响应消息框。不过，用户可以切换到其它的应用程序中。因此，为什么不给用户提供一个机会，在关闭使用定时器的其它应用程序之后，再成功地装入用户应用程序呢？此时代

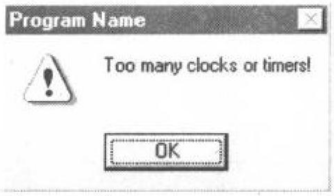


图 7-1 用于“友好终止”的消息框

码如下所示：

```
while (!SetTimer (hwnd, 1, 1000, NULL))
    if (IDCANCEL == MessageBox (hwnd,
            "Too many clocks or timers!",
            "Program Name",
            MB_ICONEXCLAMATION | MB_RETRYCANCEL))
        return FALSE;
```

这个消息框如图 7-2 所示，它有 Retry 和 Cancel 两个按钮。如果用户选择 Cancel ，那么 MessageBox 函数返回一个等于 IDCANCEL 的值，程序终止；如果用户选择 Retry ，那么 SetTimer 再次被调用。

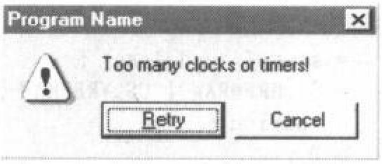


图 7-2 提供一种选择的消息框

7.2.1.2 示例程序

图 7-3 中是一个使用定时器的示例程序，这个称为 BEEPER1 的程序把定时器的时间间隔设置为 1 秒。当收到 WM_TIMER 消息时，它把客户区的颜色在红蓝之间切换，并调用 MessageBeep 函数发出蜂鸣声。虽然 MessageBeep 是 MessageBox 的相关函数，但是它经常用于产生各种蜂鸣声。在安装有声卡的 PC 机上，你可以通过使用 MessageBeep 中的各种 MB_ICON 参数得到用户选择的各种不同的声音。

BEEPER1 在 WinMain 中设置定时器，在 WndProc 窗口过程中处理 WM_TIMER 消息。在处理 WM_TIMER 消息期间， BEEPER1 调用 MessageBeep ，翻转 bFlipFlop 的值，并使窗口无效以便产生 WM_PAINT 消息。在处理 WM_PAINT 消息期间， BEEPER1 通过 GetClientRect 获取存放窗口尺寸的 RECT 结构，并通过调用 FillRect 对窗口着色。

BEEPER1.MAK

```
#-----------------------
# BEEPER1.MAK make file
#-----------------------

beeper1.exe : beeper1.obj
     $(LINKER) $(GUIFLAGS) -OUT:beeper1.exe beeper1.obj $(GUILIBS)
```

```
beeper1.obj : beeper1.c
     $(CC) $(CFLAGS) beeper1.c
```

BEEPER1.C

```
/*---------------------------------------
   BEEPER1.C  -- Timer Demo Program No. 1
                 (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Beeper1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Beeper1 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     while (!SetTimer (hwnd, ID_TIMER, 1000, NULL))
          if (IDCANCEL == MessageBox (hwnd,
                              "Too many clocks or timers!", szAppName,
                              MB_ICONEXCLAMATION | MB_RETRYCANCEL))
               return FALSE ;

     ShowWindow (hwnd, iCmdShow) ;
```

```
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL fFlipFlop = FALSE ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rc ;

     switch (iMsg)
          {
          case WM_TIMER :
               MessageBeep (0) ;

               fFlipFlop = !fFlipFlop ;
               InvalidateRect (hwnd, NULL, FALSE) ;

               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rc) ;

               hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) :
                                                      RGB(0,0,255)) ;
               FillRect (hdc, &rc, hBrush) ;
               EndPaint (hwnd, &ps) ;
               DeleteObject (hBrush) ;
               return 0 ;

          case WM_DESTROY :
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 7-3 BEEPER1程序

因为 BEEPER1 通过发声来指示它接收到了一条 WM_TIMER 消息，所以你也许会想到如果装入 BEEPER1 并在 Windows 中实现其它一些活动，就可以很好地理解 WM_TIMER

消息的不稳定性。例如，移动或者缩放 BEEPER1 窗口，将导致程序进入“模态消息循环”。Windows 通过在内部消息循环而非你程序的消息循环中捕获所有消息来禁止对移动或者缩放操作的任何干扰。通过此循环到达程序窗口的大多数消息都被丢弃，这就是 BEEPER 停止蜂鸣的原因。当完成了移动与缩放之后，你将会注意到 BEEPER1 不能取得它所丢弃的所有 WM_TIMER 消息，尽管前两条消息的间隔可能少于 1 秒。

7.2.2 方法二

设置定时器的第一种方法是把 WM_TIMER 消息发送到通常的窗口过程，而第二种方法是让 Windows 直接将定时器消息发送给你程序的另一个函数。

接收这些定时器消息的函数被称为“回调”函数，这是一个在你的程序之中但是由 Windows 调用的函数。你先告诉 Windows 此函数的地址，然后 Windows 调用此函数。这看起来也很熟悉，因为程序的窗口过程实际上也是一类回调函数。当注册窗口类时，要将函数的地址告诉 Windows，当发送消息给程序时，Windows 调用此函数。

SetTimer 并非是唯一使用回调函数的 Windows 函数。CreateDialog 和 DialogBox 函数（将在第 14 章中介绍）使用回调函数处理对话框中的消息；有几个 Windows 函数（EnumChildWindows、EnumFonts、EnumObjects、EnumProps 和 EnumWindows）把枚举信息传递给回调函数；还有几个不那么常用的函数（GrayString、LineDDA 和 SetWindowsHookEx）也要求回调函数。

像窗口过程一样，回调函数也必须定义为 CALLBACK，因为它是由 Windows 从程序的代码段调用的。回调函数的参数和回调函数的返回值依赖于回调函数的目的。同定时器有关的回调函数中，输入参数同窗口过程的输入参数一样。定时器回调函数不向 Windows 返回值。

我们把以下的回调函数称为 TimerProc（你可以用任何你喜欢的名字来称呼它），它只处理 WM_TIMER 消息。

```
VOID CALLBACK TimerProc (HWND hwnd, UINT iMsg, UINT iTimerID,
                                                 DWORD dwTime)
     {
     [处理 WM_TIMER 消息]
     }
```

输入参数 hwnd 是在调用 SetTimer 时指定的窗口句柄。Windows 只把 WM_TIMER 消息送给 TimerProc，因此 iMsg 参数总是等于 WM_TIMER。iTimerID 值是定时器 ID，dwTime 值是系统时间。

前面我已经提到过，用第一种方法设置定时器时要求下面格式的 SetTimer 调用：

```
SetTimer (hwnd, iTimerID, iMsecInterval, NULL) ;
```

当你使用回调函数处理 WM_TIMER 消息时，SetTimer 的第四个参数由回调函数的地址取代，如下所示：

```
SetTimer (hwnd, iTimerID, iMsecInterval, (TIMERPROC)TimerProc) ;
```

7.2.2.1 示例程序

让我们来看看一些示例代码，这样你就会了解这些东西是如何组合在一起的。在功能上，除了 Windows 发送一个定时器消息给 TimerProc 而非 WndProc 之外，图 7-4 所示的 BEEPER2 程序与 BEEPER1 是相同的。注意，TimerProc 和 WndProc 一起被声明在程序的开始处。

BEEPER2.MAK

```
#-----------------------
# BEEPER2.MAK make file
#-----------------------

beeper2.exe : beeper2.obj
     $(LINKER) $(GUIFLAGS) -OUT:beeper2.exe beeper2.obj $(GUILIBS)

beeper2.obj : beeper2.c
     $(CC) $(CFLAGS) beeper2.c
```

BEEPER2.C

```
/*----------------------------------------
   BEEPER2.C -- Timer Demo Program No. 2
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

#define ID_TIMER    1

LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
VOID    CALLBACK TimerProc (HWND, UINT, UINT,   DWORD ) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Beeper2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
```

```
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Beeper2 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     while (!SetTimer (hwnd, ID_TIMER, 1000, (TIMERPROC) TimerProc))
          if (IDCANCEL == MessageBox (hwnd,
                              "Too many clocks or timers!", szAppName,
                              MB_ICONEXCLAMATION | MB_RETRYCANCEL))
               return FALSE ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_DESTROY :
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

VOID CALLBACK TimerProc (HWND hwnd, UINT iMsg, UINT iTimerID, DWORD dwTime)
     {
     static BOOL fFlipFlop = FALSE ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     RECT        rc ;

     MessageBeep (0) ;
     fFlipFlop = !fFlipFlop ;

     GetClientRect (hwnd, &rc) ;

     hdc = GetDC (hwnd) ;
     hBrush = CreateSolidBrush (fFlipFlop ? RGB (255,0,0) : RGB (0,0,255)) ;

     FillRect (hdc, &rc, hBrush) ;
     ReleaseDC (hwnd, hdc) ;
```

```
DeleteObject (hBrush) ;
}
```

图 7-4 BEEPER2程序

7.2.3 方法三

设置定时器的第三种方法类似于第二种方法，只是传递给 SetTimer 的 hwnd 参数被设置为 NULL ，并且第二个参数（通常为定时器 ID ）被忽略了，最后，此函数返回定时器 ID：

```
iTimerID = SetTimer (NULL, 0, wMsecInterval, (TIMERPROC)TimerProc) ;
```

如果没有可用的定时器，那么从 SetTimer 返回的 iTimerID 值将为 NULL 。

KillTimer 的第一个参数(通常是窗口句柄)也必须为 NULL ，定时器 ID 必须是 SetTimer 的返回值：

```
KillTimer (NULL, iTimerID) ;
```

传递给 TimerProc 定时器函数的 hwnd 参数也必须是 NULL 。

这种设置定时器的方法很少被使用。如果在你的程序中有一系列不同时刻的 SetTimer 调用，而又不希望跟踪你已经用过了哪些定时器 ID ，那么使用此方法是很方便的。

既然你已经知道了如何使用 Windows 定时器，就可以开始讨论一组有用的定时器程序了。

7.3 定时器用于时钟

时钟是定时器最明显的应用，因此让我们来看看两个时钟，一个模拟时钟，一个数字时钟。图 7-5 中所示的 DIGCLOCK 程序，创建一个弹出式窗口，位于显示器右下角，此程序显示星期、日期和时间，如图 7-6 中所示。

DIGCLOCK.MAK

```
#-----------------------
# DIGCLOCK.MAK make file
#-----------------------

digclock.exe : digclock.obj
     $(LINKER) $(GUIFLAGS) -OUT:digclock.exe digclock.obj $(GUILIBS)

digclock.obj : digclock.c
     $(CC) $(CFLAGS) digclock.c
```

DIGCLOCK.C

```
/*--------------------------------------
   DIGCLOCK.C -- Digital Clock Program
                 (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <time.h>

#define ID_TIMER    1

#define YEAR  (datetime->tm_year % 100)
#define MONTH (datetime->tm_mon  + 1)
#define MDAY  (datetime->tm_mday)
#define WDAY  (datetime->tm_wday)
#define HOUR  (datetime->tm_hour)
#define MIN   (datetime->tm_min)
#define SEC   (datetime->tm_sec)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
void SizeTheWindow (int *, int *, int *, int *) ;

char  sDate[2], sTime[2], sAMPM[2][5] ;
int   iDate, iTime ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "DigClock" ;
     HWND        hwnd ;
     MSG         msg ;
     int         xStart, yStart, xClient, yClient ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = NULL ;

     RegisterClassEx (&wndclass) ;

     SizeTheWindow (&xStart, &yStart, &xClient, &yClient) ;

     hwnd = CreateWindow (szAppName, szAppName,
                          WS_POPUP | WS_DLGFRAME | WS_SYSMENU,
                          xStart,  yStart,
                          xClient, yClient,
                          NULL, NULL, hInstance, NULL) ;
```

```
     if (!SetTimer (hwnd, ID_TIMER, 1000, NULL))
          {
          MessageBox (hwnd, "Too many clocks or timers!", szAppName,
                      MB_ICONEXCLAMATION | MB_OK) ;
          return FALSE ;
          }

     ShowWindow (hwnd, SW_SHOWNOACTIVATE) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void SizeTheWindow (int *pxStart,  int *pyStart,
                    int *pxClient, int *pyClient)
     {
     HDC        hdc ;
     TEXTMETRIC tm ;

     hdc = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
     GetTextMetrics (hdc, &tm) ;
     DeleteDC (hdc) ;

     *pxClient = 2 * GetSystemMetrics (SM_CXDLGFRAME) + 16*tm.tmAveCharWidth ;
     *pxStart  =     GetSystemMetrics (SM_CXSCREEN)   - *pxClient ;
     *pyClient = 2 * GetSystemMetrics (SM_CYDLGFRAME) + 2*tm.tmHeight ;
     *pyStart  =     0 ;
     }

void SetInternational (void)
     {
     static char cName[] = "intl" ;

     iDate = GetProfileInt (cName, "iDate", 0) ;
     iTime = GetProfileInt (cName, "iTime", 0) ;

     GetProfileString (cName, "sDate",  "/", sDate,    2) ;
     GetProfileString (cName, "sTime",  ":", sTime,    2) ;
     GetProfileString (cName, "s1159", "AM", sAMPM[0], 5) ;
     GetProfileString (cName, "s2359", "PM", sAMPM[1], 5) ;
     }

void WndPaint (HWND hwnd, HDC hdc)
     {
     static char szWday[] = "Sun\0Mon\0Tue\0Wed\0Thu\0Fri\0Sat" ;
     char        cBuffer[40] ;
     int         iLength ;
     RECT        rect ;
     struct tm  *datetime ;
```

```
     time_t      lTime ;

     time (&lTime) ;
     datetime = localtime (&lTime) ;

     iLength = wsprintf (cBuffer, "  %s  %d%s%02d%s%02d  \r\n",
               (PSTR) szWday + 4 * WDAY,
               iDate == 1 ? MDAY  : iDate == 2 ? YEAR  : MONTH, (PSTR) sDate,
               iDate == 1 ? MONTH : iDate == 2 ? MONTH : MDAY,  (PSTR) sDate,
               iDate == 1 ? YEAR  : iDate == 2 ? MDAY  : YEAR) ;

     if (iTime == 1)
          iLength += wsprintf (cBuffer + iLength, "  %02d%s%02d%s%02d  ",
                              HOUR, (PSTR) sTime, MIN, (PSTR) sTime, SEC) ;
     else
          iLength += wsprintf (cBuffer + iLength, "  %d%s%02d%s%02d %s  ",
                              (HOUR % 12) ? (HOUR % 12) : 12,
                              (PSTR) sTime, MIN, (PSTR) sTime, SEC,
                              (PSTR) sAMPM [HOUR / 12]) ;

     GetClientRect (hwnd, &rect) ;
     DrawText (hdc, cBuffer, -1, &rect, DT_CENTER | DT_NOCLIP) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;

     switch (iMsg)
          {
          case WM_CREATE :
               SetInternational () ;
               return 0 ;

          case WM_TIMER :
               InvalidateRect (hwnd, NULL, FALSE) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               WndPaint (hwnd, hdc) ;
               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_WININICHANGE :
               SetInternational () ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case WM_DESTROY :
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
```

```
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 7-5 DIGCLOCK程序

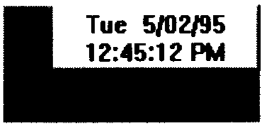


图 7-6　DIGCLOCK窗口

到目前为止所有的程序都把 CreateWindow 函数的第三个参数窗口风格设置为 WS_OVERLAPPEDWINDOW 。DIGCLOCK 使用窗口风格：

```
WS_POPUP | WS_DLGFRAME | WS_SYSMENU
```

这就创建了一个带有对话框和系统菜单的称为“弹出式”的窗口风格。弹出风格通常用于对话框和消息框中，应用程序很少使用。DIGCLOCK 还使用了 ShowWindow 调用的另一变体：

```
ShowWindow (hwnd, SW_SHOWNOACTIVATE) ;
```

通常情况下，运行一个程序时该程序就称为活动窗口。SW_NOSHOWACTIVATE 告知 Windows ，装入 DIGCLOCK 的那个程序应该一直保持为活动窗口。不过，在该窗口上按鼠标键或者按 Alt-Tab 或 Alt-Esc 也能使 DIGCLOCK 成为活动窗口。虽然 DIGCLOCK 没有系统菜单框，但是当 DIGCLOCK 活动时按 Alt-Spacebar 仍然可以访问系统菜单。如果选择 Move ，那么你可以使用键盘来移动窗口。

7.3.1　弹出式窗口的定位和缩放

DIGCLOCK 弹出式窗口定位在显示器的右下角。窗口必须能够容纳 2 行，每行 16 个字符。DIGCLOCK.C 中的 SizeTheWindow 过程确定在 CreateWindow 调用中使用的正确参数。一般情况下，程序在第一次创建窗口之前不能获得文本的大小，因为它需要使用窗口句柄来得到设备环境句柄。DIGCLOCK 用 CreateIC 来获得屏幕的信息设备环境句柄以解决此问题，这个函数类似于通常用来为打印机创建设备环境的 CreateDC （我将在第 15 章中演示），不过它通常是用于从设备环境中获取信息。文本尺寸与从 GetSystemMetrics 中得到的信息结合起来就可以推导出最初的起始位置和窗口尺寸。

7.3.2　获取日期和时间

在 WndProc 函数中，DIGCLOCK 使用 C 函数 time 和 localtime 来决定当前日期和时间。localtime 函数把我们需要的所有信息都放入一个结构中，在程序开始处定义的几个宏使 wsprintf 调用具有更强的可读性(你应该避免在你的 Windows 程序中使用 MS-DOS 或 ROM BIOS 函数调用，而使用 Windows 函数或者 C 运行库）。

7.3.3 国际化

Windows 包括国际化支持。在 Windows 安装期间创建的 WIN.INI 文件中有一个以[intl]开头的段，它列出了日期、时间、货币和数字的格式。你可以使用以下三种格式来显示日期：月-日-年、年-月-日或者日-月-年，这三个数字间的分隔符可以是斜杠、短横、点号或者其它任意字符。你也可以使用 12 小时或者 24 小时的格式来显示时间，时、分、秒之间通常用冒号或者点号分开。

DIGCLOCK 中的 SetInternational 函数使用 Windows 的 GetProfileInt （对整数）和 GetProfileString（对字符串）函数从 WIN.INI 文件取得格式信息，这些调用必须含有 Windows 不能在 WIN.INI 中找到所需值时的缺省值。SetInternational 在一个全局变量中保存这个值，这个全局变量与 WIN.INI 中的字符串同名。 WndPaint 函数使用 WIN.INI 获得的值来格式化日期与时间的显示，然后调用 DrawText 在窗口中居中显示这两行文本。

每当 DIGCLOCK 的窗口过程收到一个 WM_TIMER 消息时，它就使窗口无效从而产生一个 WM_PAINT 消息。但是 WndProc 在接收到 WM_WININICHANGE 消息时也使窗口无效。任意一个修改 WIN.INI 的应用程序都把 WM_WININICHANGE 消息发送给所有活动窗口。如果 WIN.INI 的[intl]部分被修改了，那么 DIGCLOCK 就会接收到通知并得到新的国际信息。运行 DIGCLOCK 和 Windows 的控制面板程序，选择国际化图标，然后修改日期格式、日期分隔符、时间格式或者时间分隔符，你就可以看到这种修改所带来的变化。按下 Enter ，控制面板更新 WIN.INI 文件， DIGCLOCK 的显示反映了这种变化——Windows 消息的神奇起作用了。

当窗口过程收到 WM_WININICHANGE 消息时，它使用：

```
InvalidateRect (hwnd, NULL, TRUE) ;
```

使窗口无效。当 DIGCLOCK 接收到 WM_TIMER 消息时，它使用：

```
InvalidateRect (hwnd, NULL, FALSE) ;
```

使窗口无效。最后一个参数为 TRUE 时，它告诉 Windows 在绘制窗口之前擦除背景色；为 FALSE 时，告诉 Windows 简单地在当前背景上绘制窗口。处理 WM_TIMER 消息时我们使用 FALSE ，因为这种方法可以减少显示器的闪烁。你可能奇怪为什么还要用 TRUE 值。

当处理 WM_WININICHANGE 时使用 TRUE 是必要的，因为如果把时间格式从 12 小时改变为 24 小时，那么所显示的字符串的长度会发生变化。而 WM_TIMER 消息导致的最大变化是两个字符——如当 12/31/95 前进一天到 1/1/96 时，为了解决在长度与比例字体方面的变化引起的问题，在 WndPaint 显示的格式化字符串两头留有空格。

我们也可以让 DIGCLOCK 处理 WM_TIMECHANGE 消息，这样它将系统时间与日期发生变化这一消息通知应用程序。 DIGCLOCK 因 WM_TIMER 消息而每秒更新一次，实际上没有必要这么作，对 WM_TIMECHANGE 消息的处理使得每分钟更新一次的时钟变得更为合理。

7.3.4 构造模拟时钟

模拟时钟不必关心国际化问题，但是由于图形所引起的复杂性却抵销了这种简化。为了正确地产生一个模拟时钟，你需要知道如何使用映射模式乃至一些三角函数。ANACLOCK 程序如图 7-7 所示，它的屏幕显示如图 7-8 所示。

ANACLOCK.MAK

```
#------------------------
# ANACLOCK.MAK make file
#------------------------

anaclock.exe : anaclock.obj
     $(LINKER) $(GUIFLAGS) -OUT:anaclock.exe anaclock.obj $(GUILIBS)

anaclock.obj : anaclock.c
     $(CC) $(CFLAGS) anaclock.c
```

ANACLOCK.C

```
/*----------------------------------------
   ANACLOCK.C -- Analog Clock Program
                 (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <string.h>
#include <time.h>
#include <math.h>

#define ID_TIMER    1
#define TWOPI       (2 * 3.14159)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "AnaClock" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
```

```
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = NULL ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Analog Clock",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     if (!SetTimer (hwnd, ID_TIMER, 1000, NULL))
          {
          MessageBox (hwnd, "Too many clocks or timers!", szAppName,
                      MB_ICONEXCLAMATION | MB_OK) ;
          return FALSE ;
          }

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void SetIsotropic (HDC hdc, int cxClient, int cyClient)
     {
     SetMapMode (hdc, MM_ISOTROPIC) ;
     SetWindowExtEx (hdc, 1000, 1000, NULL) ;
     SetViewportExtEx (hdc, cxClient / 2, -cyClient / 2, NULL) ;
     SetViewportOrgEx (hdc, cxClient / 2,  cyClient / 2, NULL) ;
     }

void RotatePoint (POINT pt[], int iNum, int iAngle)
     {
     int   i ;
     POINT ptTemp ;

     for (i = 0 ; i < iNum ; i++)
          {
          ptTemp.x = (int) (pt[i].x * cos (TWOPI * iAngle / 360) +
                            pt[i].y * sin (TWOPI * iAngle / 360)) ;

          ptTemp.y = (int) (pt[i].y * cos (TWOPI * iAngle / 360) -
                            pt[i].x * sin (TWOPI * iAngle / 360)) ;

          pt[i] = ptTemp ;
          }
     }
```

```
void DrawClock (HDC hdc)
     {
     int   iAngle ;
     POINT pt[3] ;

     for (iAngle = 0 ; iAngle < 360 ; iAngle += 6)
          {
          pt[0].x =   0 ;
          pt[0].y = 900 ;

          RotatePoint (pt, 1, iAngle) ;

          pt[2].x = pt[2].y = iAngle % 5 ? 33 : 100 ;

          pt[0].x -= pt[2].x / 2 ;
          pt[0].y -= pt[2].y / 2 ;

          pt[1].x  = pt[0].x + pt[2].x ;
          pt[1].y  = pt[0].y + pt[2].y ;

          SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;

          Ellipse (hdc, pt[0].x, pt[0].y, pt[1].x, pt[1].y) ;
          }
     }

void DrawHands (HDC hdc, struct tm *datetime, BOOL bChange)
     {
     static POINT pt[3][5] = { 0, -150, 100, 0, 0, 600, -100, 0, 0, -150,
                               0, -200,  50, 0, 0, 800,  -50, 0, 0, -200,
                               0,    0,   0, 0, 0,   0,    0, 0, 0,  800 } ;
     int          i, iAngle[3] ;
     POINT        ptTemp[3][5] ;

     iAngle[0] = (datetime->tm_hour * 30) % 360 + datetime->tm_min / 2 ;
     iAngle[1] =  datetime->tm_min  *  6 ;
     iAngle[2] =  datetime->tm_sec  *  6 ;

     memcpy (ptTemp, pt, sizeof (pt)) ;

     for (i = bChange ? 0 : 2 ; i < 3 ; i++)
          {
          RotatePoint (ptTemp[i], 5, iAngle[i]) ;

          Polyline (hdc, ptTemp[i], 5) ;
          }
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int        cxClient, cyClient ;
     static struct tm  dtPrevious ;
     BOOL              bChange ;
     HDC               hdc ;
     PAINTSTRUCT       ps ;
```

```
     time_t           lTime ;
     struct tm       *datetime ;

     switch (iMsg)
          {
          case WM_CREATE :
               time (&lTime) ;
               datetime = localtime (&lTime) ;

               dtPrevious = * datetime ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_TIMER :
               time (&lTime) ;
               datetime = localtime (&lTime) ;

               bChange = datetime->tm_hour != dtPrevious.tm_hour ||
                         datetime->tm_min  != dtPrevious.tm_min ;
               hdc = GetDC (hwnd) ;

               SetIsotropic (hdc, cxClient, cyClient) ;

               SelectObject (hdc, GetStockObject (WHITE_PEN)) ;
               DrawHands (hdc, &dtPrevious, bChange) ;

               SelectObject (hdc, GetStockObject (BLACK_PEN)) ;
               DrawHands (hdc, datetime, TRUE) ;

               ReleaseDC (hwnd, hdc) ;

               dtPrevious = *datetime ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               SetIsotropic (hdc, cxClient, cyClient) ;
               DrawClock    (hdc) ;
               DrawHands    (hdc, &dtPrevious, TRUE) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               KillTimer (hwnd, ID_TIMER) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 7-7 ANACLOCK程序

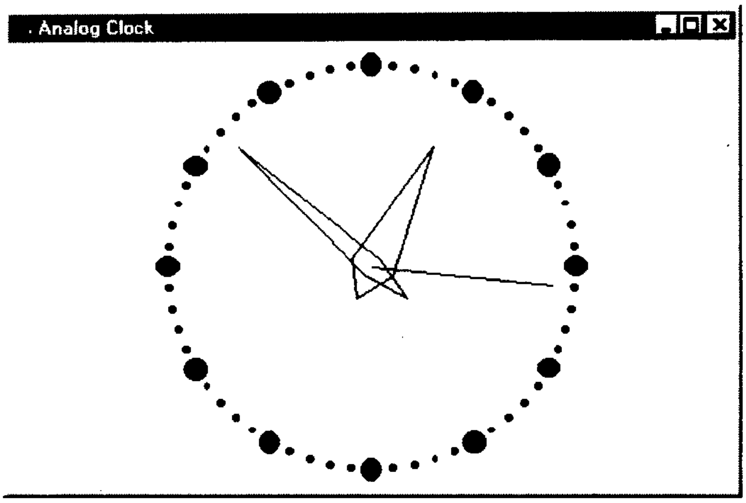


图 7-8 ANACLOCK窗口

各向同性(isotropic)映射模式对于这样的应用来说是理想的，ANACLOCK.C 中的 SetIsotropic 函数负责设置此模式。在调用 SetMapMode 之后，SetIsotropic 将窗口范围设置为 1000，并将视口范围设置为客户区的一半宽度和客户区的负的一半高度。视口原点被设置为客户区的中心。我在第 4 章中讨论过，这将创建一个笛卡儿坐标系，其点(0, 0)位于客户区的中心，在所有方向上的范围都是 1000。

RotatePoint 函数是用到三角的地方，此函数的三个参数分别是一个或者多个点的数组、数组中点的个数以及以度为单位的旋转角度。函数以原点为中心按顺时针方向（这对一个时钟正合适）旋转这些点。例如，如果传给函数的点是(0, 100)——即 12:00 的位置——而角度为 90 度，那么该点将被变换为(100, 0)——即 3:00。它使用下列公式来做到这一点：

```
x' = x * cos (a) + y * sin (a) ;
y' = y * cos (a) - x * sin (a) ;
```

RotatinPosition 函数在绘制时钟表面的点和表针时都是有用的，我们将马上看到这一点。

DrawClock 函数绘制 60 个时钟表面的点，从顶部(12:00)开始，其中每个点离原点 900 单位，因此第一个点位于(0, 900)，此后的每个点按顺时针依次增加 6 度。这些点中的 12 个直径为 100 个单位；其余的为 33 个单位。使用 Ellipse 函数来画点。

DrawHands 函数绘制时钟的时针、分针和秒针。定义表针轮廓（当它们垂直向上时的形状）的坐标存放在一个 POINT 结构的数组中。根据时间，这些坐标使用 RotatePoint 函数进行旋转，并用 Windows 的 Polyline 函数进行显示。注意时针和分针只有当传递给 DrawHands 的 bChange 参数为 TRUE 时才被显示。当程序更新时钟的表针时，在多数情况下时针和分针不需要重画。

现在让我们将注意力转到窗口过程。在 WM_CREATE 消息期间，窗口过程获取当前时间并将它存放在名为 dtPrevious 的变量中，这个变量将在以后被用于确定时针或者分针从上次更新以来是否改变过。

第一次绘制时钟是在第一个 WM_PAINT 消息期间，这只不过是依次调用 SetIsotropic、

DrawClock 和 DrawHands，后者的 bChange 参数被设置为 TRUE。

在 WM_TIMER 消息期间，WndProc 首先获取新的时间并确定是否需要重新绘制时针和分针。如果需要，则使用一个白色画笔和上一次的时间绘制所有的表针，从而有效地擦除它们。否则，只对秒针使用白色画笔进行擦除，然后，再使用一个黑色画笔绘制所有的表针。

7.4 Windows 标准时间

如果你正在阅读 Windows 函数的参考手册，那么你很可能对 DIGCLOCK 中不使用 Windows 的 GetCurrentTime 函数感到奇怪。其实，GetCurrentTime 取得的是“Windows 时间”而非实际的时间，是从本次 Windows 会话开始之后所经历的时间(以毫秒为单位)。GetCurrentTime 经常用于计算与 GetMessageTime 返回的时间的差值。在处理一条消息时，可以通过这两个调用来确定它在消息队列中直到处理时已经存留了多长时间。

7.5 动　　画

在第 4 章中，我们介绍了一些使用位图和内存设备环境的技术。因为显示小位图非常快，所以你可以将位图（和 Windows 定时器一起）用于一些简单的动画。

是的，该是编写撞球程序的时候了。如图 7-9 中所示的 BOUNCE 程序构造了一个在窗口的客户区中跳动的球，球本身是一个位图。程序首先通过创建位图来创建这个球，然后将它选入一个内存设备环境中，然后再进行简单的 GDI 函数调用。程序使用从另一个内存设备环境中的位块传输，在显示器上绘制位图化的球。

BOUNCE.MAK

```
#----------------------
# BOUNCE.MAK make file
#----------------------

bounce.exe : bounce.obj
     $(LINKER) $(GUIFLAGS) -OUT:bounce.exe bounce.obj $(GUILIBS)

bounce.obj : bounce.c
     $(CC) $(CFLAGS) bounce.c
```

BOUNCE.C

```
/*---------------------------------------
   BOUNCE.C - Bouncing Ball Program
              (c) Charles Petzold, 1996
 ---------------------------------------*/

#include <windows.h>
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Bounce" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;
```

Figure 7-9. The BOUNCE program.

```
     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Bouncing Ball",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     if (!SetTimer (hwnd, 1, 50, NULL))
          {
          MessageBox (hwnd, "Too many clocks or timers!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK) ;
          return FALSE ;
          }

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HBITMAP hBitmap ;
     static int     cxClient, cyClient, xCenter, yCenter, cxTotal, cyTotal,
                    cxRadius, cyRadius, cxMove, cyMove, xPixel, yPixel ;
     HBRUSH         hBrush ;
     HDC            hdc, hdcMem ;
     int            iScale ;
```

```
switch (iMsg)
     {
     case WM_CREATE :
          hdc = GetDC (hwnd) ;
          xPixel = GetDeviceCaps (hdc, ASPECTX) ;
          yPixel = GetDeviceCaps (hdc, ASPECTY) ;
          ReleaseDC (hwnd, hdc) ;
          return 0 ;

     case WM_SIZE :
          xCenter = (cxClient = LOWORD (lParam)) / 2 ;
          yCenter = (cyClient = HIWORD (lParam)) / 2 ;

          iScale = min (cxClient * xPixel, cyClient * yPixel) / 16 ;

          cxRadius = iScale / xPixel ;
          cyRadius = iScale / yPixel ;

          cxMove = max (1, cxRadius / 2) ;
          cyMove = max (1, cyRadius / 2) ;

          cxTotal = 2 * (cxRadius + cxMove) ;
          cyTotal = 2 * (cyRadius + cyMove) ;

          if (hBitmap)
               DeleteObject (hBitmap) ;

          hdc = GetDC (hwnd) ;
          hdcMem = CreateCompatibleDC (hdc) ;
          hBitmap = CreateCompatibleBitmap (hdc, cxTotal, cyTotal) ;
          ReleaseDC (hwnd, hdc) ;

          SelectObject (hdcMem, hBitmap) ;
          Rectangle (hdcMem, -1, -1, cxTotal + 1, cyTotal + 1) ;

          hBrush = CreateHatchBrush (HS_DIAGCROSS, 0L) ;
          SelectObject (hdcMem, hBrush) ;
          SetBkColor (hdcMem, RGB (255, 0, 255)) ;
          Ellipse (hdcMem, cxMove, cyMove, cxTotal - cxMove,
                                           cyTotal - cyMove) ;
          DeleteDC (hdcMem) ;
          DeleteObject (hBrush) ;
          return 0 ;

     case WM_TIMER :
          if (!hBitmap)
               break ;

          hdc = GetDC (hwnd) ;
          hdcMem = CreateCompatibleDC (hdc) ;
          SelectObject (hdcMem, hBitmap) ;

          BitBlt (hdc, xCenter - cxTotal / 2,
                       yCenter - cyTotal / 2, cxTotal, cyTotal,
                  hdcMem, 0, 0, SRCCOPY) ;

          ReleaseDC (hwnd, hdc) ;
          DeleteDC (hdcMem) ;

          xCenter += cxMove ;
```

```
            yCenter += cyMove ;

            if ((xCenter + cxRadius >= cxClient) ||
                (xCenter - cxRadius <= 0))
                    cxMove = -cxMove ;

            if ((yCenter + cyRadius >= cyClient) ||
                (yCenter - cyRadius <= 0))
                    cyMove = -cyMove ;
            return 0 ;

        case WM_DESTROY :
            if (hBitmap)
                 DeleteObject (hBitmap) ;

            KillTimer (hwnd, 1) ;
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 7-9　BOUNCE程序

每当程序得到一个 WM_SIZE 消息， BOUNCE 就重构这个球。这要求一个和视频显示兼容的内存设备环境：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

球的周长为客户区高的十六分之一或者宽的十六分之一，选择其中较小的一个。然而，程序构造一个比球要大的位图：在它的四边中的每一条边，位图比球多出其半径的一半。

```
hBitmap = CreateCompatibleBitmap (hdc, cxTotal, cyTotal) ;
```

在位图被选入一个内存设备环境后，这个位图被着色为白色并用作背景：

```
Rectangle (hdcMem, -1, -1, xTotal + 1, yTotal + 1) ;
```

一个对角条纹画笔被选入内存设备环境中，并在位图的中心绘制球：

```
Ellipse (hdcMem, xMove, yMove, xTotal - xMove, yTotal - yMove) ;
```

当球被移动时，球四周的边空有效地擦除了球的上一个图象。在另一个位置重新绘制球只需要以 SRCCOPY 为 ROP 码来调用 BitBlt：

```
BitBlt (hdc, xCenter - xTotal / 2,
             yCenter - yTotal / 2, xTotal, yTotal,
             hdcMem, 0, 0, SRCCOPY) ;
```

BOUNCE 演示了在显示器上移动图象的最简单方法，但是这个方法并不能满足一般的目的。如果你对动画感兴趣，那么你将希望使用其它一些 ROP 码（诸如 SRCINVERT 等）在源和目的上执行一种异或操作。动画的其它技术涉及 Windows 调色板（以及 AnimatePalette 函数）和 CreateDIBSection 函数。

第 8 章　子窗口控制

第六章的 CHECKER 系列给出了显示矩形网格的程序。当你在一个矩形中按下鼠标键时，该程序就写下一个 x ；如果你再按一次鼠标键，那么 x 就消失。虽然这个程序的 CHECKER1 和 CHECKER2 版本只使用一个主窗口，但 CHECKER3 版本却为每个矩形使用一个子窗口。这些矩形由一个叫做 ChildWndProc 的独立窗口过程维护。

如果有必要，无论矩形是否被选中，那么都可以给 ChildWndProc 增加一种向其父窗口过程（ WndProc ）发送消息的手段。通过调用 GetParent ，子窗口过程能确定它的父窗口的窗口句柄：

```
hwndParent = GetParent (hwnd) ;
```

其中， hwnd 是子窗口的窗口句柄。然后它可以向其父窗口过程发送消息：

```
SendMessage (hwndPareng, iMsg, wParam, lParam) ;
```

那么 iMsg 应该设置为什么呢？你可以随意地设置，只要数字值位于范围 WM_USER 到 0x7FFF 之间，这些数字代表和预定义的 WM_message 不冲突的消息。也许对这条消息，子窗口可以将 wParam 设置为它的子窗口 ID 。如果该子窗口已经被画上 x ，那么 lParam 可以被置为 1 ；如果该子窗口未被画上 x ，那么 lParam 被置为 0 。这是一种选择。

事实上，这是在建立一个“子窗口控制”。当子窗口的状态已经改变时，子窗口处理鼠标和键盘消息并通知父窗口。使用这种方法，子窗口就变成了其父窗口的输入设备。它将与自己在屏幕上的图形外观相应的处理、它对用户输入的响应以及在发生一个重要的输入事件时通知另一个窗口的方法封装起来。

虽然你可以建立自己的子窗口控制，但是也可以利用一些预定义的窗口类（和窗口过程）来建立标准的子窗口控制，你一定在别的基于 Windows 的程序中见过了这些控制。这些控制采用的形式有：按钮、选择框、编辑框、列表框、组合框、文本串和滚动条。例如，如果想在你的电子表格程序的某个角上放置一个标有“ Recalculate（重新计算）”的按钮，那么你可以通过调用 CreateWindow 来创建这个按钮。你不必担心鼠标操作、按钮显示操作或当按下该按钮时的自动闪烁操作，这些是由 Windows 内部完成的。你所要做的只是捕获 WM_COMMAND 消息——当按钮被按下时，它通过这一消息通知你的窗口过程。

真的这样简单吗？是的，一点不错。

子窗口控制最常用在对话框中。在第 11 章中你将会看到，子窗口控制的位置和尺寸是在示例程序的资源描述文本的对话框模板中定义的。但是，你也可以使用预定义的、位于普通窗口客户区表面的子窗口控制。你可以调用一次 CreateWindow 来建立一个子窗口，并通过调用 MoveWindow 来调整子窗口的位置和尺寸。父窗口过程向子窗口控制发送消息，

子窗口控制向父窗口过程返回消息。

在建立普通窗口时，首先定义窗口类，并使用 RegisterClass 将其注册到 Windows 中，然后用 CreateWindow 命令在该窗口类的基础上建立一个普通窗口，从第 2 章开始，我们就是这么做的。但是，当你使用预先定义的某个控制时，不必为子窗口注册窗口类，窗口类已经存在于 Windosw 之中，并且有一个如下的名字：“ button ”、“ static ”、“ scrollbar ”、“ edit ”、“ listbox ”或者“ combobox ”。你只需在 CreateWindow 中把它们用作窗口类参数。CreateWindow 中的窗口风格参数准确地定义了子窗口控制的外形和功能。Windows 包含了处理发送给基于这些窗口类的子窗口的消息的窗口过程。

直接在你的窗口表面使用子窗口控制完成某些任务，这些任务的层次低于在对话框中使用子窗口控制所要求的层次。这里，对话框管理器在你的程序和控制之间增加一个隔离层。值得指出的是，你可能会发现在你的窗口表面创建的子窗口控制没有利用 Tab 键或光标移动键将输入焦点从一个控制移动到另一个控制的内部功能。子窗口控制能够获得输入焦点，但是获得后，它将不能把输入焦点返回给父窗口。这就是本章要解决的问题。

8.1 按钮类

下面我们将通过一个叫做 BTNLOOK(“ button look ”)的程序来开始介绍按钮窗口类，如图 8-1 所示。BTNLOOK 创建 10 个子窗口按钮控制，每个控制对应于一个标准的按钮风格，因此共有 10 种标准按钮风格。

BTNLOOK.MAK

```
#----------------------
# BTNLOOK.MAK make file
#----------------------

btnlook.exe : btnlook.obj
     $(LINKER) $(GUIFLAGS) -OUT:btnlook.exe btnlook.obj $(GUILIBS)

btnlook.obj : btnlook.c
     $(CC) $(CFLAGS) btnlook.c
```

BTNLOOK.C

```
/*----------------------------------------
   BTNLOOK.C -- Button Look Program
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>

struct
     {
     long style ;
     char *text ;
     }
```

```
     button[] =
     {
     BS_PUSHBUTTON,          "PUSHBUTTON",
     BS_DEFPUSHBUTTON,       "DEFPUSHBUTTON",
     BS_CHECKBOX,            "CHECKBOX",
     BS_AUTOCHECKBOX,        "AUTOCHECKBOX",
     BS_RADIOBUTTON,         "RADIOBUTTON",
     BS_3STATE,              "3STATE",
     BS_AUTO3STATE,          "AUTO3STATE",
     BS_GROUPBOX,            "GROUPBOX",
     BS_AUTORADIOBUTTON,     "AUTORADIO",
     BS_OWNERDRAW,           "OWNERDRAW"
     } ;

#define NUM (sizeof button / sizeof button[0])

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BtnLook" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Button Look",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

```
     return msg.wParam ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szTop[]    = "iMsg            wParam       lParam",
                 szUnd[]    = "----            ------       ------",
                 szFormat[] = "%-16s%04X-%04X    %04X-%04X",
                 szBuffer[50] ;
     static HWND hwndButton[NUM] ;
     static RECT rect ;
     static int  cxChar, cyChar ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     int         i ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
               ReleaseDC (hwnd, hdc) ;

               for (i = 0 ; i < NUM ; i++)
                    hwndButton[i] = CreateWindow ("button", button[i].text,
                         WS_CHILD | WS_VISIBLE | button[i].style,
                         cxChar, cyChar * (1 + 2 * i),
                         20 * cxChar, 7 * cyChar / 4,
                         hwnd, (HMENU) i,
                         ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
               return 0 ;

          case WM_SIZE :
               rect.left   = 24 * cxChar ;
               rect.top    =  2 * cyChar ;
               rect.right  = LOWORD (lParam) ;
               rect.bottom = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               InvalidateRect (hwnd, &rect, TRUE) ;

               hdc = BeginPaint (hwnd, &ps) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

               SetBkMode (hdc, TRANSPARENT) ;

               TextOut (hdc, 24 * cxChar, cyChar, szTop, sizeof (szTop) - 1) ;
               TextOut (hdc, 24 * cxChar, cyChar, szUnd, sizeof (szUnd) - 1) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;
```

```
          case WM_DRAWITEM :
          case WM_COMMAND :
               ScrollWindow (hwnd, 0, -cyChar, &rect, &rect) ;

               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

               TextOut (hdc, 24 * cxChar, cyChar * (rect.bottom /
                        cyChar - 1),
                        szBuffer,
                        wsprintf (szBuffer, szFormat,
                        iMsg == WM_DRAWITEM ? "WM_DRAWITEM" : "WM_COMMAND",
                        HIWORD (wParam), LOWORD (wParam),
                        HIWORD (lParam), LOWORD (lParam))) ;

               ReleaseDC (hwnd, hdc) ;
               ValidateRect (hwnd, &rect) ;

               break ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 8-1 BTNLOOK程序

当你按动按钮时，按钮就发送一个 WM_COMMAND 消息给父窗口过程，也就是我们所熟悉的 WndProc 。 BTNLOOK 的 WndProc 将该消息的 wParam 参数和 lParam 参数显示在客户区的右边，如图 8-2 中所示。

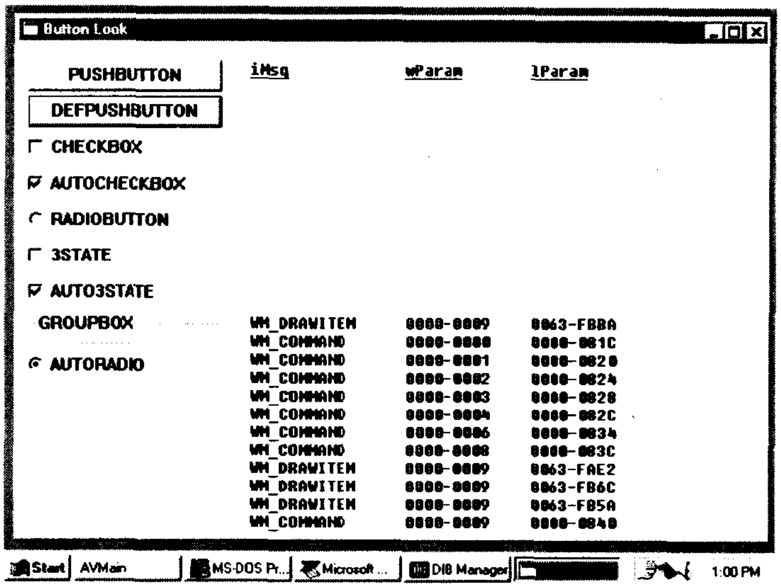


图 8-2 BTNLOOK的显示

具有风格 BS_OWNERDRAW 的按钮在窗口上显示为一个背景阴影，因为这种风格的按钮是由程序来负责绘制的。该按钮表示它需要由包含 lParam 消息参数的 WM_DRAWITEM 消息来绘制，而 lParam 消息参数是一个指向 DRAWITEMSTRUCT 类型结构的指针。在 BNTLOOK 中，这些消息也被同样显示。（我将在这一章的后面更详细地讨论这种拥有者绘制按钮。）

8.1.1　创建子窗口

BTNLOOK 定义了一个叫做按钮（ button ）的结构，它包含了按钮窗口风格和描述性文本串，它们对应于 10 个按钮类型。所有按钮窗口风格的前两个字母都是 BS ，它表示 "button style"（按钮风格）。

10 个按钮子窗口是在 WndProc 中处理 WM_CREATE 消息的过程中使用一个 for 循环建立的。 CreateWindow 调用使用下面这些参数：

Class Name （类名）	“ button ”
Window text （窗口文本）	button[i].text
Window style （窗口风格）	WS_CHILD I WS_VISIBLE I button[i].style
x position （ x 位置）	cxChar
y position （ y 位置）	cyChar(1+2*i)
Width （宽度）	20 * cxChar
Height （高度）	7 * cyChar / 4
Parent window （父窗口）	hwnd
Child window ID （子窗口 ID ）	(HMENU)i
Instance handle （实例句柄）	((LPCREATESTRUCT)lParam) -> hInstance
Extra parameters （附加参数）	NULL

类名参数是预定义的名字；窗口风格使用 WS_CHILD 与 WS_VISIBLE 同“按钮”结构中的 10 个按钮风格（ BS_PUSHBUTTON 、 BS_DEFPUSHBUTTON 等）之一的组合；窗口文本参数（对普通窗口来说，它是显示在标题行中的文本）将在每个按钮上显示出来。我简单地使用标识按钮风格的文本。

x 位置和 y 位置两个参数说明子窗口左上角的位置，该位置的相应原点是父窗口客户区的左上角；宽度和高度参数规定每个子窗口的宽度和高度。

对每个子窗口，它的子窗口 ID 参数应该各不相同。在处理来自子窗口的 WM_COMMAND 消息时， ID 帮助你的窗口过程识别出相应的子窗口。注意子窗口 ID 是作为 CreateWindow 的一个参数传递的，该参数通常用于指定程序的菜单，因此子窗口 ID 必须被强制转换为 HMENU 。

CreateWindow 调用的实例句柄看起来有点奇怪，但是它利用了如下事实，即在处理 WM_CREATE 消息的过程中， lParam 实际上是指向 CREATESTRUCT （“创建结构”）结构的指针，该 CREATESTRUCT 有一个 hInstance 成员。所以将 lParam 转换成指向 CREATESTRUCT 结构的一个指针，并取出 hInstance 。

（有些 Windows 程序使用名为 hInst 的全局变量，使窗口过程能访问 WinMain 中的实例句柄。在 WinMain 中，你只需在建立主窗口之前设置：

```
hInst = hInstance;
```

在第 6 章中的 CHECKER3 程序中，我们曾用 GetWindowLong 来得到实例句柄：

```
GetWindowLong (hwnd, GWL_HINSTANCE) ;
```

这几种方法都是正确的。）

在调用 CreateWindow 之后，我们不必再为这些子窗口做任何事情，由 Windows 中的按钮窗口过程负责维护它们，并处理所有重画工作。（BS_OWNERDRAW 风格的按钮例外，它要求程序绘制它，这些将在后面加以讨论。）在程序终止时，如果父窗口已经被清除，那么 Windows 将清除这些子窗口。

8.1.2 子窗口向父窗口发消息

当你运行 BTNLOOK 时，将看到在客户区的左边会显示出不同的按钮类型。我在前面已经提到过，当用鼠标单击按钮时，子窗口控制就向其父窗口发送一个 WM_COMMAND 消息。BTNLOOK 捕获 WM_COMMAND 消息并显示 wParam 和 lParam 的值，它们的含义如下：

LOWORD (wParam)	子窗口 ID
HIWORD (wParam)	通知码
lParam	子窗口句柄

如果你正在将 16 位 Windows 程序移植过来，那么注意改变这些消息参数以容纳 32 位的句柄。

子窗口 ID 是在创建子窗口时传递给 CreateWindow 的值。在 BTNLOOK 中，这些 ID 被显示在客户区中，并使用 0 到 9 分别标识 10 个按钮。子窗口句柄是 Windows 从 CreateWindow 返回的值。

通知码是子窗口用来更详细地告诉其父窗口该消息含义的子消息码。按钮通知码的可能值在 Windows 头文件中定义如下：

按钮通知码标识符	值
BN_CLICKED	0
BN_PAINT	1
BN_HILITE	2
BN_UNHILITE	3
BN_DISABLE	4
BN_DOUBLECLICKED	5

从 1 到 5 的通知码是用于一种叫做 BS_USERBUTTON 的已不再使用的按钮的，因此你将

只能看到 BN_CLICKED 码。

你会注意到，当用鼠标单击按钮时，该按钮文本的周围会有虚线。这表示该按钮拥有了输入焦点，所有键盘输入都将传送给子窗口按钮控制，而不是传送给主窗口。但是，当该按钮控制拥有输入焦点时，它将忽略所有的键盘输入，但空格键例外，此时空格键与鼠标具有相同的效果。

8.1.3　父窗口向子窗口发送消息

虽然 BTNLOOK 中没有演示这一事实，但是父窗口过程也能向子窗口控制发送消息。在 Windows 头文件中定义了按钮特有的五条消息，每条消息的开头字母都是"BM"，它代表"button message（按钮消息）"。这些消息是：

```
BM_GETCHECK
BM_SETCHECK
BM_GETSTATE
BM_SETSTATE
BM_SETSTYLE
```

BM_GETCHECK 和 BM_SETCHECK 消息由父窗口发送给子窗口控制，以获取或者设置复选框和单选按钮的选中标记。BM_GETSTATE 和 BM_SETSTATE 消息表示按钮处于正常状态还是（鼠标或空格键按下时的）"按下"状态。（我们将在讨论按钮的每种类型时，看到这些消息是如何起作用的。）BM_SETSTYLE 消息允许你在按钮创建之后改变按钮风格。

每个子窗口控制都具有一个在其兄弟中唯一的窗口句柄和 ID 值。对于句柄和 ID 这两者，知道其中的一个你就可以获得另一个。如果你知道子窗口控制的窗口句柄，那么你可以用：

```
id = GetWindowLong (hwndChild, GWL_ID) ;
```

来获得 ID。

第 6 章中的 CHECKER3 程序表明窗口可以将数据保存在注册窗口类时所保留的一个特殊区域中。保存子窗口 ID 的区域是在创建子窗口时由 Windows 保留的。你也可以使用：

```
id = GetDlgCtrlID (hwndChild) ;
```

虽然函数名的"Dlg"部分指的是对话框，但实际上这是一个通用的函数。

知道 ID，你也可以获得子窗口句柄：

```
hwndChild = GetDlgItem (hwndParent, id) ;
```

8.1.4　压入按钮

在 BTNLOOK 中显示的前两个按钮是"压入"按钮。按钮是一个矩形，包含了 CreateWindow 调用中窗口文本参数所指定的文本。该矩形占用了在 CreateWindow 或者

MoveWindow 调用中给出的全部高度和宽度，而文本在矩形的中心。

压入按钮控制主要用来触发一个立即响应的动作，而不保留任何形式的开/关指示。两种类型的按钮控制有两种窗口风格，分别叫做 BS_PUSHBUTTON 和 BS_DEFPUSHBUTTON，BS_DEFPUSHBUTTON 中的"DEF"代表"default"（缺省）。当用来设计对话框时，BS_PUSHBUTTON 控制和 BS_DEFPUSHBUTTON 控制的作用不同。但是当用作子窗口控制时，两种类型的按钮作用相同，尽管 BS_DEFPUSHBUTTON 的边框要粗一些。

当按钮的高度为 SYSTEM_FONT 字符高度的 7/4 倍时，按钮的外观最好，其中 SYSTEM_FONT 字符由 BTNLOOK 使用；按钮的宽度至少调节到文本的宽度再加上两个字符的宽度。

当鼠标光标在按钮中时，按下鼠标键将使按钮用三维阴影重画自己，就好像真的被按下了一样。放开鼠标键时，就恢复按钮的原样，并向父窗口发送一个 WM_COMMAND 消息和 BN_CLICKED 。与其它按钮类型相似，当按钮拥有输入焦点时，在文本的周围有虚线，按下和释放空格键与按下和释放鼠标键具有相同的效果。

你可以通过给窗口发送 BM_SETSTATE 消息来模拟按钮闪动。如下操作导致按钮被按下：

```
SendMessage (hwnButton, BM_SETSTATE, 1, 0) ;
```

下面的调用使按钮恢复正常：

```
SendMessage (hwndButton, BM_SETSTATE, 0, 0) ;
```

hwndButton 窗口句柄是从 CreateWindow 调用返回的值。

你也可以向按钮发送 BM_GETSTATE 消息，子窗口控制返回按钮的当前状态——如果按钮被按下，则返回 TRUE ；如果按钮处于正常状态，则返回 FALSE 。但是，绝大多数应用不需要这一信息。因为按钮不保留任何开/关信息，所以 BM_SETCHECK 消息和 BM_GETCHECK 消息不会被用到。

8.1.5 复选框

复选框是一个文本框，文本通常出现在复选框的右边。（如果你在创建按钮时指定了 BS_LEFTTEXT 风格，那么文本会出现在左边。）复选框通常用于允许用户对选项进行选择的应用程序中。复选框的常用功能如同一个开关：按动复选框一次，就出现 X ，再按一次则 X 消失。

复选框最常用的两种风格是 BS_CHECKBOX 和 BS_AUTOCHECKBOX 。在使用 BS_CHECKBOX 时，你需要自己向该控制发送 BM_SETCHECK 消息来设置 X 标记。wParam 参数置 1 时设置 X 标记，置 0 时清除 X 标记。通过向该控制发送 BM_GETCHECK 消息，你可以得到该复选框的当前状态。在处理来自控制的 WM_COMMAND 消息时，你可以用如下的指令来翻转 X 标记：

```
SendMessage ((HWND) lParam, BM_SETCHECK, (WPARAM)
        !SendMessage ((HWND) lParam, BM_GETCHECK, 0, 0), 0) ;
```

注意第二个 SendMessage 调用前面的操作符“！”，其中 lParam 的低字是在 WM_COMMAND 消息中传给用户窗口过程的子窗口句柄。如果你以后又想知道按钮的状态，那么可以向它发送另一条 BM_GETCHECK 消息；你也可以将当前状态保存在你的窗口过程中的一个静态变量里，或者向它发送 BM_SETCHECK 消息来初始化带 X 的 BS_CHECKBOX 复选框：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对 BS_AUTOCHECKBOX 风格，按钮自己触发 X 标记的开和关，你的窗口过程可以忽略 WM_COMMAND 消息。当你需要按钮的当前状态时，可以向控制发送 BM_GETCHECK 消息：

```
iCheck = (int)SendMessage (hwndButton, BM_GETCHECK, 0, 0) ;
```

如果该按钮被选中，则 iCheck 的值为 TRUE 或者非零数；如果按钮未被选中，则 iCheck 的值为 FALSE 或者 0 。

其余两种复选框风格是 BS_3STATE 和 BS_AUTO3STATE，正如它们的名字所暗示的，这两种风格能显示第三种状态——复选框内是灰色——它出现在向控制发送 wParam=2 的 WM_SETCHECK 消息时。灰色是向用户表明此框是不能被选中的，即它是被禁用的。但是，当你按下它时，该复选框仍然会向其父窗口发送消息。使一个复选框被禁用的较好方法将在后面加以描述。

复选框沿矩形的左边框对齐，并集中在调用 CreateWindow 过程中规定的矩形的顶边和底边之间，在该矩形内的任何地方按下鼠标都会向其父窗口发送一条 WM_COMMAND 消息。复选框的最小高度是一个字符的高度，最小宽度是文本中的字符数加 2 。

8.1.6　单选按钮

单选按钮的形状是一个圆圈，而不是方框，除此之外，它非常象复选框。圆圈内的加重圆点表示该单选按钮已经被选中。单选按钮有窗口风格 BS_RADIOBUTTON 和 BS_AUTORADIOBUTTON 两种，但是后者只用于对话框。在对话框中，单选按钮组常常用来表示相互排斥的选项。与复选框不同，单选按钮的工作与开关不一样，当第二次按单选按钮时，它的状态保持不变。

当你收到来自单选按钮的 WM_COMMAND 消息时，应该向它发送 wParam 等于 1 的 BM_SETCHECK 消息来显示其选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 1, 0) ;
```

对相同组中的其它所有单选按钮，你可以通过向它们发送 wParam 等于 0 的 BM_SETCHECK 消息来显示其未选中状态：

```
SendMessage (hwndButton, BM_SETCHECK, 0, 0) ;
```

8.1.7　分组框

分组框即风格为 BS_GROUPBOX 的选择框，它是按钮类中的特例，既不处理鼠标输

入和键盘输入，也不向其父窗口发送消息。分组框是一个矩形框，窗口文本在其顶部显示。分组框常用来包含其它的按钮控制。

8.1.8 改变按钮文本

你可以通过 SetWindowText 来改变按钮（或者其它任何窗口）内的文本：

```
SetWindowText (hwnd, pszString) ;
```

其中 hwnd 是要改变的窗口的句柄，pszString 是一个指向 0 终结串的指针。对于一般的窗口来说，这个文本是标题行的文本；对于按钮控制来说，它是随该按钮显示的文本。

你也可以获取窗口的当前文本：

```
iLength = GetWindowText (hwnd, pszBuffer, iMaxLength) ;
```

iMaxLength 指定拷贝到 pszBuffer 指向的缓冲区中的最大字符数，该函数返回拷贝的字符数。你可以首先通过下面的调用来获得特定文本的长度：

```
iLength = GetWindowTextLength (hwnd) ;
```

8.1.9 可见的和启用的按钮

为了接收鼠标和键盘输入，子窗口必须是可见的（被显示）和被启用的。当窗口是可见的而未被启用时，那么窗口以灰色而非黑色显示正文。

如果在建立子窗口时，你没有将 WS_VISIBLE 包含在窗口类中，那么直到调用 ShowWindow 时子窗口才被显示出来：

```
ShowWindow (hwndChild, SW_SHOWNORMAL) ;
```

如果你将 WS_VISIBLE 包含在窗口类中，就没有必要调用 ShowWindow 。但是，你可以通过调用 ShowWindow 将子窗口隐蔽起来：

```
ShowWindow (hwndChild, SW_HIDE) ;
```

你可以通过下面的调用来确定子窗口是否可见：

```
IsWindowVisible (hwndChild) ;
```

你也可以使子窗口被启用或者不被启用。缺省情况下，窗口是被启用的。你可以通过下面的调用使窗口不被启用：

```
EnableWindow (hwndChild, FALSE) ;
```

对按钮控制，这具有使按钮文本串变灰的作用。按钮将不再对鼠标输入和键盘输入做出响应，这是表示按钮选项当前不可用的最好方法。

你可以通过下面的调用使子窗口再次被启用：

```
EnableWindow (hwndChild, TRUE) ;
```

你还可以使用下面的调用来确定子窗口是否被启用:

```
IsWindowEnabled (hwndChild) ;
```

8.1.10 按钮和输入焦点

我在本章前面已经提到过，当用鼠标单击按钮、复选框、单选框和拥有者绘制按钮时，它们接收到输入焦点。这些控制使用文本周围的虚线来表示它拥有了输入焦点。当子窗口控制得到输入焦点时，其父窗口就失去了输入焦点；所有的键盘输入都进入子窗口控制中，而不会进入父窗口中。但是，子窗口控制只对空格键作出响应，此时空格键的作用就如同鼠标键一样。这种情形导致了一个明显的问题：你的程序失去了对键盘处理的控制。让我们看看对此我们能做一些什么。

我在第 5 章中已经提到过，当 Windows 将输入焦点从一个窗口（例如一个父窗口）转换到另一个窗口（例如一个子窗口控制）时，它首先给正在失去输入焦点的窗口发送一条 WM_KILLFOCUS 消息， wParam 参数是接受输入焦点的窗口的句柄。然后， Windows 向正在接受输入焦点的窗口发送一条 WM_SETFOCUS 消息，同时 wParam 是正在失去输入焦点的窗口的句柄。（在这两种情况下， wParam 可能取值 NULL，它表示没有窗口拥有或者正在接收输入焦点。）

通过处理 WM_KILLFOCUS 消息，父窗口可以阻止子窗口控制获得输入焦点。假定数组 hwndChild 包含了所有子窗口的窗口句柄。（它们是在调用 CreateWindow 来建立窗口的时候保存到数组中的。） NUM 是子窗口的数目：

```
case WM_KILLFOCUS:
    for (I=0; i<NUM; I++)
        if (hwndChild[i]== (HWND)wParam)
            {
            SetFocus (hwnd) ;
            break;
            }
    return 0;
```

在这个代码中，当父窗口获知它正在失去输入焦点而让它的某个子窗口得到输入焦点时，它将调用 SetFocus 来恢复输入焦点给自己所有。

下面是达到相同目的的更简单（但不太直观）的方法：

```
case WM_KILLFOCUS:
    if (hwnd == GetParent (wParam))
        SetFocus (hwnd) ;
    return 0;
```

但是，这两种方法都有缺点：它们阻止按钮对空格键作出响应，因为该按钮总是得不

到输入焦点。一个更好的方法是使按钮得到输入焦点，也能让用户用 Tab 键从一个按钮转移到另一个按钮。这听起来似乎不可能，但是在本章的后面，我们将要说明在 COLORS1 程序中如何用“窗口子类化”技术来实现这种方法。

8.2 控制与颜色

你可以在图 8-2 中看到，许多按钮的显示看起来并不正确。压入按钮还好，但是在其它按钮却带有一个本不应该在那里的一个矩形灰色背景。这是因为这些按钮本来是为对话框中的显示而设计的，而在 Windows 95 中，对话框有一个灰色的表面。我们的窗口有一个白色的表面，这是因为我们在 WNDCLASS 结构中就是这样定义的。

```
wndclass.hbrBackgroup = (HBRUSH)GetStockObject (WHITE_BRUSH) ;
```

我们已经这么做了，因为我们经常在客户区中显示文本，而 GDI 使用在缺省设备环境中定义的文本颜色和背景颜色，它们总是黑色和白色。为了使这些按钮更加美观一些，我们必须要么改变客户区的颜色使之和按钮的背景颜色一致，要么以某种方法将按钮的背景颜色改变为白色。

第一步是理解 Windows 对“系统颜色”的使用。

8.2.1 系统颜色

Windows 保留了 25 种系统颜色以供各种显示使用。你可以使用 GetSysColor 和 SetSysColor 来获得和设置这些颜色。在 Windows 头文件中定义的标识符规定了系统颜色。使用 SetSysColor 设置的系统颜色只在当前 Windows 会话过程中有效。

借助 Windows 的控制面板程序或者通过修改 WIN.INI 文件的[colors]段，你可以改变系统颜色。在[colors]段中，对 25 种系统颜色（与 GetSysColor 和 SetSysColor 的标识符不同）使用了关键字，25 种系统颜色的后面跟着红、绿、蓝三种颜色的值，该值的变化范围是 0 到 255 。下面的表格说明了这 25 种系统颜色是如何在 GetSysColor 、 SetSysColor 以及 WIN.INI 关键字中用常数来标识的。这张表是按 COLOR_常数值（以 0 开始到 24 结束）顺序排列的：

GetSysColor 和 SetSysColor	WIN.INI
COLOR_SCROLLBAR	Scrollbar
COLOR_BACKGROUND	Background
COLOR_ACTIVECAPTION	ActiveTitle
COLOR_INACTIVECAPTION	InactiveTitle
COLOR_MENU	Menu
COLOR_WINDOW	Window
COLOR_WINDOWFRAME	WindowFrame

续表

GetSysColor 和 SetSysColor	WIN.INI
COLOR_MENUTEXT	MenuText
COLOR_WINDOWTEXT	WindowText
COLOR_CAPTIONTEXT	TitleText
COLOR_ACTIVEBORDER	ActiveBorder
COLOR_INACTIVEBORDER	InactiveBorder
COLOR_APPWORKSPACE	AppWorkspace
COLOR_HIGHLIGHT	Hilight
COLOR_HIGHLIGHTTEXT	HilightText
COLOR_BTNFACE	ButtonFace
COLOR_BTNSHADOW	ButtonShadow
COLOR_GRAYTEXT	GrayText
COLOR_BTNTEXT	ButtonText
COLOR_INACTIVECAPTIONTEXT	InactiveTitleText
COLOR_BTNHIGHLIGHT	ButtonHilight
COLOR_3DDKSHADOW	ButtonDkShadow
COLOR_3DLIGHT	ButtonLight
COLOR_INFOTEXT	InfoText
COLOR_INFOBK	InfoWindow

这 25 种颜色的缺省值是由显示驱动程序提供的。Windows 一直使用这些缺省值，除非被 WIN.INI 的[colors]段中的设置所修改，而这可以通过控制面板来进行。

坏消息：虽然这些颜色中的许多似乎都是不言自明的（例如，COLOR_BACKGROUND 是所有窗口后面的桌面区域的颜色），但是在 Windows 95 中系统颜色的使用变得非常混乱。以前，Windows 在视觉上要比今天简单得多，随着看起来越来越难于控制的三维外观的使用，相应地也需要越来越多的系统颜色。

8.2.2 按钮颜色

对按钮来说这个问题更加明显：COLOR_BTNFACE 被用于压入按钮主要的表面颜色以及其它按钮主要的背景颜色。COLOR_BTNSHADOW 被建议用作压入按钮右下边以及复选框内部和单选按钮圆点的阴影。对于压入按钮，COLOR_BTNTEXT 被用作文本颜色；而对于其它的按钮，则使用 COLOR_WINDOWTEXT 作为文本颜色。还有其它几种系统颜色用于按钮设计的各个部分。

因此，如果你想在我们的客户区表面显示按钮，那么一种避免颜色冲突的方法是屈服于这些系统颜色。首先，在定义窗口类时使用 COLOR_BTNFACE 作为你的客户区的背景颜色：

```
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE+1) ;
```

你可以在 BTNLOOK 程序中尝试这种方法。当 WNDCLASSEX 结构中的 hbrBackground 值是这个值时， Windows 会明白这实际上指的是一种系统颜色而非一个实际的句柄。Windows 要求当你在 WNDCLASSEX 结构的 hbrBackground 域中指定这些标识符时加 1 ，这样做的目的是防止其值为 NULL ，而没有任何其它目的。如果你的程序运行中，系统颜色恰好发生了变化，那么客户区将变得无效， Windows 将使用新的 COLOR_BTNFACE 值。

但是现在我们又引发了另一个问题。当你使用 TextOut 显示文本时， Windows 使用的是在设备环境中为背景颜色（它擦除文本后的背景）和文本颜色定义的值，其缺省值为白色（背景）和黑色（文本），而不管系统颜色和窗口类结构中的 hbrBackground 域为何值。所以，你需要使用 SetTextColor 和 SetBkColor 将文本和文本背景的颜色改变为系统颜色。你可以在获得设备环境句柄之后这么做：

```
SetBkColor (hdc, GetSysColor (COLOR_BTNFACE)) ;
SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT)) ;
```

这样，客户区背景、文本背景和文本的颜色都与按钮的颜色一致了。

8.2.3 WM_CTLCOLORBTN 消息

我们已经看到了我们是如何将客户区的颜色和文本颜色调节成按钮的背景颜色的。我们是否可以将程序中按钮的颜色调节为我们喜欢的颜色呢？理论上没问题，但在实践中别这样做。

用 SetSysColor 来改变按钮的外观可能不是你想做的，这会影响当前在 Windows 下运行的所有程序；也是用户不太喜欢的。

更好的方法（同样也只是理论上的）是处理 WM_CTLCOLORBTN 消息，这是当子窗口即将给其客户区着色时，由按钮控制发送给其父窗口过程的一个消息。父窗口可以利用这个机会来改变子窗口过程将要用来着色的颜色。（在 Windows 的 16 位版本中，一个称为 WM_CTLCOLOR 的消息被用于所有的控制，现在针对每种类型的标准控制，分别代之以不同的消息。）

当父窗口过程收到 WM_CTLCOLORBTN 消息时， wParam 消息参数是按钮的设备环境句柄， lParam 是按钮的窗口句柄。当父窗口过程得到这个消息时，按钮控制已经获得了它的设备环境。当你的窗口过程处理一个 WM_CTLCOLORBTN 消息时，你必须完成如下三个动作：

- 使用 SetTextColor 有选择地设置一种文本颜色。
- 使用 SetBkColor 有选择地设置一种文本背景颜色。
- 将一个画刷句柄返回给子窗口。

理论上，子窗口使用该画刷来着色背景。当不再需要这个画刷时，你应该负责清除它。

下面是使用 WM_CTLCOLORBTN 的问题所在：只有压入按钮和拥有者绘制按钮才给其父窗口发送 WM_CTLCOLORBTN ，而只有拥有者绘制按钮才响应父窗口过程对消息的

处理，使用画刷来着色背景。这基本上是没有意义的，因为无论怎样都是由父窗口来负责绘制拥有者绘制按钮。

在本章后面，我们将说明，在某些情况下，一些类似于 WM_CTLCOLORBTN 但适用于其它类型控制的消息更为有用。

8.2.4　拥有者绘制按钮(Owner-Drawn Buttons)

如果你想对按钮的全部可见部分实行全面的控制，而不想被键盘逻辑所干扰，那么你可以使用 BS_OWNERDRAW 来建立一个按钮，如图 8-3 中的 OWNERDRAW 程序：

OWNERDRW.MAK

```
#------------------------
# OWNERDRW.MAK make file
#------------------------

ownerdrw.exe : ownerdrw.obj
     $(LINKER) $(GUIFLAGS) -OUT:ownerdrw.exe ownerdrw.obj $(GUILIBS)

ownerdrw.obj : ownerdrw.c
     $(CC) $(CFLAGS) ownerdrw.c
```

OWNERDRW.C

```
/*----------------------------------------------
   OWNERDRW.C -- Owner-Draw Button Demo Program
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>

#define IDC_SMALLER      1
#define IDC_LARGER       2

#define BTN_WIDTH        (8 * cxChar)
#define BTN_HEIGHT       (4 * cyChar)

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "OwnerDrw" ;
     MSG         msg ;
     HWND        hwnd ;
     WNDCLASSEX  wndclass ;

     hInst = hInstance ;
```

```
     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Owner-Draw Button Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void Triangle (HDC hdc, POINT pt[])
     {
     SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
     Polygon (hdc, pt, 3) ;
     SelectObject (hdc, GetStockObject (WHITE_BRUSH)) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND       hwndSmaller, hwndLarger ;
     static int        cxClient, cyClient, cxChar, cyChar ;
     int               cx, cy ;
     LPDRAWITEMSTRUCT pdis ;
     POINT             pt[3] ;
     RECT              rc ;

     switch (iMsg)
          {
          case WM_CREATE :
               cxChar = LOWORD (GetDialogBaseUnits ()) ;
               cyChar = HIWORD (GetDialogBaseUnits ()) ;

                                   // Create the owner-draw pushbuttons
```

```
          hwndSmaller = CreateWindow ("button", "",
                              WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                              0, 0, BTN_WIDTH, BTN_HEIGHT,
                              hwnd, (HMENU) IDC_SMALLER, hInst, NULL) ;

          hwndLarger = CreateWindow ("button", "",
                              WS_CHILD | WS_VISIBLE | BS_OWNERDRAW,
                              0, 0, BTN_WIDTH, BTN_HEIGHT,
                              hwnd, (HMENU) IDC_LARGER, hInst, NULL) ;
          return 0 ;

     case WM_SIZE :
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;
                    // Move the buttons to the new center

          MoveWindow (hwndSmaller, cxClient / 2 - 3 * BTN_WIDTH  / 2,
                                   cyClient / 2 -     BTN_HEIGHT / 2,
                                   BTN_WIDTH, BTN_HEIGHT, TRUE) ;

          MoveWindow (hwndLarger,  cxClient / 2 +     BTN_WIDTH  / 2,
                                   cyClient / 2 -     BTN_HEIGHT / 2,
                                   BTN_WIDTH, BTN_HEIGHT, TRUE) ;
          return 0 ;

     case WM_COMMAND :
          GetWindowRect (hwnd, &rc) ;

                    // Make the window 10% smaller or larger

          switch (wParam)
               {
               case IDC_SMALLER :
                    rc.left   += cxClient / 20 ;
                    rc.right  -= cxClient / 20 ;
                    rc.top    += cyClient / 20 ;
                    rc.bottom -= cyClient / 20 ;

                    break ;

               case IDC_LARGER :
                    rc.left   -= cxClient / 20 ;
                    rc.right  += cxClient / 20 ;
                    rc.top    -= cyClient / 20 ;
                    rc.bottom += cyClient / 20 ;

                    break ;
               }

          MoveWindow (hwnd, rc.left, rc.top, rc.right - rc.left,
                                             rc.bottom - rc.top, TRUE) ;
          return 0 ;

     case WM_DRAWITEM :
```

```
pdis = (LPDRAWITEMSTRUCT) lParam ;

          // Fill area with white and frame it black

FillRect (pdis->hDC, &pdis->rcItem,
          (HBRUSH) GetStockObject (WHITE_BRUSH)) ;
FrameRect (pdis->hDC, &pdis->rcItem,
           (HBRUSH) GetStockObject (BLACK_BRUSH)) ;

          // Draw inward and outward black triangles

cx = pdis->rcItem.right  - pdis->rcItem.left ;
cy = pdis->rcItem.bottom - pdis->rcItem.top  ;

switch (pdis->CtlID)
     {
     case IDC_SMALLER :
          pt[0].x = 3 * cx / 8 ;  pt[0].y = 1 * cy / 8 ;
          pt[1].x = 5 * cx / 8 ;  pt[1].y = 1 * cy / 8 ;
          pt[2].x = 4 * cx / 8 ;  pt[2].y = 3 * cy / 8 ;

          Triangle (pdis->hDC, pt) ;

          pt[0].x = 7 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
          pt[1].x = 7 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
          pt[2].x = 5 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

          Triangle (pdis->hDC, pt) ;

          pt[0].x = 5 * cx / 8 ;  pt[0].y = 7 * cy / 8 ;
          pt[1].x = 3 * cx / 8 ;  pt[1].y = 7 * cy / 8 ;
          pt[2].x = 4 * cx / 8 ;  pt[2].y = 5 * cy / 8 ;

          Triangle (pdis->hDC, pt) ;

          pt[0].x = 1 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
          pt[1].x = 1 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
          pt[2].x = 3 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

          Triangle (pdis->hDC, pt) ;

          break ;

     case IDC_LARGER :

          pt[0].x = 5 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
          pt[1].x = 3 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
          pt[2].x = 4 * cx / 8 ;  pt[2].y = 1 * cy / 8 ;

          Triangle (pdis->hDC, pt) ;

          pt[0].x = 5 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
          pt[1].x = 5 * cx / 8 ;  pt[1].y = 3 * cy / 8 ;
          pt[2].x = 7 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;
```

```
                    Triangle (pdis->hDC, pt) ;

                    pt[0].x = 3 * cx / 8 ;  pt[0].y = 5 * cy / 8 ;
                    pt[1].x = 5 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                    pt[2].x = 4 * cx / 8 ;  pt[2].y = 7 * cy / 8 ;

                    Triangle (pdis->hDC, pt) ;

                    pt[0].x = 3 * cx / 8 ;  pt[0].y = 3 * cy / 8 ;
                    pt[1].x = 3 * cx / 8 ;  pt[1].y = 5 * cy / 8 ;
                    pt[2].x = 1 * cx / 8 ;  pt[2].y = 4 * cy / 8 ;

                    Triangle (pdis->hDC, pt) ;

                    break ;
               }

               // Invert the rectangle if the button is selected

          if (pdis->itemState & ODS_SELECTED)
               InvertRect (pdis->hDC, &pdis->rcItem) ;

               // Draw a focus rectangle if the button has the focus

          if (pdis->itemState & ODS_FOCUS)
               {
               pdis->rcItem.left   += cx / 16 ;
               pdis->rcItem.top    += cy / 16 ;
               pdis->rcItem.right  -= cx / 16 ;
               pdis->rcItem.bottom -= cy / 16 ;

               DrawFocusRect (pdis->hDC, &pdis->rcItem) ;
               }

          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 8-3 OWNERDRAW程序

该程序在其客户区的中央包含了两个按钮，如图 8-4 所示。左边的按钮有四个三角形指向按钮的中央，按下该按钮时，窗口的尺寸将缩小 10 %。右边的按钮有四个向外指的三角形，按下此按钮时，窗口的尺寸将增大 10 %。使用 BS_OWNERDRAW 按钮风格来画按钮的大多数程序通常给按钮使用较小的位图，但是，在 OWNERDRAW 程序中，只是在按钮的表面上绘制三角形。

在处理 WM_CREATE 消息期间， OWNERDRAW 通过调用 GetDialogBaseUnits 函数来获取系统字体的平均高度和宽度，通常可以很方便地用这个函数获取此信息。然后，

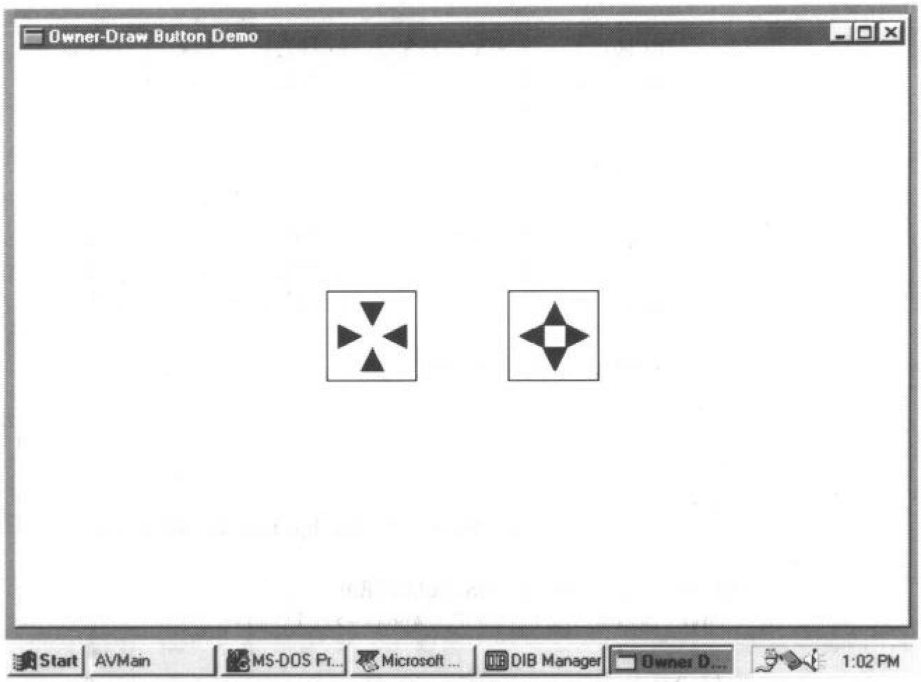


图 8-4 OWNERDRAW的显示

OWNERDRAW 建立两个 BS_OWNERDRAW 风格的按钮；按钮的宽度是系统字体的 8 倍，高度是系统字体的 4 倍。（在使用预定义好的位图绘制按钮时，这些尺寸在 VGA 上建立的按钮为 64 象素宽 64 象素高，知道这些数据非常有用。）这些按钮尚未定位，在处理 WM_SIZE 消息期间，通过调用 MoveWindow 函数，OWNERDRAW 将按钮定位在客户区的中心。

按下这些按钮时，它们就会产生 WM_COMMAND 消息。为了处理这些 WM_COMMAND 消息，OWNERDRAW 调用 GetWindowRect，将整个窗口（不只是客户区）的位置和尺寸存放在 RECT（矩形）结构中，这个位置相对于屏幕。然后，根据按下的是左边的还是右边的按钮，OWNERDRAW 调节这个矩形结构的各个域值。程序再通过调用 MoveWindow 来重新确定位置和尺寸。这将产生另一条 WM_SIZE 消息，按钮被重新定位在客户区的中央。

如果这是程序所做的全部处理，那么这完全可以，只不过按钮是不可见的。使用 BS_OWNERDRAW 风格建立的按钮在需要重新着色的任何时候都会向它的父窗口发送一条 WM_DRAWITEM 消息，这出现在以下几种情况中：当按钮被创建时，当按钮被按下或者被松开时，当按钮得到或者失去输入焦点时以及当按钮需要重新着色的任何时候。

在处理 WM_DRAWITEM 消息期间，lParam 消息参数是指向类型 DRAWITEMSTRUCT 结构的指针，OWNERDRAW 程序将这个指针保存在 pdis 变量中，这个结构包含了画该按钮时程序所必需的消息（这个结构也可以为自绘列表框和菜单项所使用）。对按钮而言非常重要的结构域有 hDC（按钮的设备环境）、rcItem（提供按钮尺寸的 RECT 结构）、CtlID（控制窗口 ID）和 itemState（它说明按钮是否被按下或者按钮是否拥有输入焦点）。

调用 FillRect 用白色画刷抹掉按钮的表面，调用 FramRect 在按钮的周围画上黑框，由

此 OWNERDRAW 便启动了 WM_DRAWITEM 处理过程。然后，通过调用 Polygon ，OWNERDRAW 在按钮上画出 4 个黑色实心的三角形。这是通常的情形。

如果按钮当前被按下，那么 DRAWITEMSTRUCT 的 itemState 域中的某位将被置位。你可以使用 ODS_SELECTED 常量来测试这些位。如果这些位被置位，那么 OWNERDRAW 将通过调用 InvertRect 将按钮翻转为相反的颜色。如果按钮拥有输入焦点，那么 itemState 的 ODS_FOCUS 位将被置位，在这种情况下， OWNERDRAW 通过调用 DrawFocusRect ，在按钮的边界内画一个虚线的矩形。

在使用拥有者绘制按钮时应该注意以下几个方面： Windows 获得设备环境并将其作为 DRAWITEMSTRUCT 结构的一个域；保持设备环境处于你找到它时所处的状态；任何被选进设备环境的 GDI 对象都必需被释放；另外，当心不要在定义按钮边界的矩形外面进行绘制。

8.3 静态类

在 CreateWindow 函数中指定窗口类为 “ static （静态）” ，你就可以建立静态的子窗口控制。这些子窗口非常 “文静” ，它既不接收鼠标或键盘输入，也不向父窗口发送 WM_COMMAND 消息（当你在静态子窗口上移动或者按下鼠标时，这个子窗口将捕获 WM_HCHITTEST 消息并将 HTTRANSPARENT 的值返回给 Windows ，这将使 Windows 向其下层窗口，通常是它的父窗口，发送相同的 WM_NCHITTEST 消息。父窗口常常将该消息传递给 DefWindowProc ，在这里，它被转换为客户区的鼠标消息）。

前六个静态窗口风格只简单地在子窗口的客户区内画一个矩形或者边框。在下表的上部， “ RECT ” 静态风格（左列）是填充矩形的风格；三个 “ FRAME ” 风格（右列）是没有填充的矩形轮廓：

SS_BLACKRECT	SS_BLACKFRAME
SS_GRAYRECT	SS_GRAYFRAME
SS_WHITERECT	SS_WHITEFRAME

“ BLACK ” 、 “ GRAY ” 、 “ WHITE ” 并不意味着黑、灰和白色，这些颜色是由系统颜色决定的，如下所示：

静态控制	系统颜色
BLACK	COLOR_3DDKSHADOW
GRAY	COLOR_BTNSHADOW
WHITE	COLOR_BTNHIGHTLIGHT

对这些风格， CreateWindow 调用中的窗口文本域被忽略。矩形的左上角开始于 x 位置坐标和 y 位置坐标，这些坐标都相对于父窗口（你也可以使用 SS_ETCHEDHORZ 、

SS_ETCHEDVERT 或者 SS_ETCHEDFRAME ，采用“灰色”和“白色”创建一个形似阴影的边框）。

静态类也包括了三种文本风格： SS_LEFT 、 SS_RIGHT 和 SS_CENTER 。它们创建左对齐、右对齐和居中文本。文本在 CreateWindow 调用的窗口文本参数中给出，并且在以后可以用 SetWindowText 来改变它。当静态控制的窗口过程显示文本时，它使用 DrawText 函数以及 DT_WORDBREAK 、 DT_NOCLIP 和 DT_EXPANDTABS 参数。文本在子窗口的矩形内可以按字进行换行。

这三种文本风格子窗口的背景通常为 COLOR_BTNFACE ，而文本本身是 COLOR_WINDOWTEXT 。在捕获 WM_CTLCOLORSTATIC 消息时，你可以通过调用 SetTextColor 来改变文本颜色，通过 SetBkColor 来改变背景颜色，并返回背景画刷句柄。后面的 COLORS1 程序演示了这一点。

最后，静态类还包括了窗口风格 SS_ICON 和 SS_USERITEM ，但是当它们被用作子窗口控制时却没有任何意义。我们在讨论对话框时还要涉及它们。

8.4 滚动条类

在第三章的 SYSTEMS 系列程序的设计中，我首次讨论了滚动条，也讨论了“窗口滚动条”和“滚动控制”之间的一些区别。 SYSTEMS 使用窗口滚动条，它出现在窗口的右边和底部。你可以在创建窗口时通过将标识符 WS_VSCROLL 、 WS_HSCROLL 或者两者都包含在窗口风格中，从而给窗口增加滚动条。现在我们准备建立一些滚动条控制，它们是能在父窗口的客户区的任何地方出现的子窗口。你可以使用预定义的窗口类“ scrollbar ”以及两个滚动条风格 SBS_VERT 和 SBS_HORZ 中的一个来建立子窗口滚动条控制。

与按钮控制（以及将在后面讨论的编辑和列表框控制）不同，滚动条控制不向父窗口发送 WM_COMMAND 消息，而是像窗口滚动条那样发送 WM_VSCROLL 和 WM_HSCROLL 消息。在处理滚动条消息时，你可以通过 lParam 参数的高位字，区分开窗口滚动条与滚动条控制，对于窗口滚动条其值为 0 ，对于滚动条控制其值为滚动条窗口句柄。对窗口滚动条和滚动条控制来说， wParam 参数和 lParam 低位字的含义是相同的。

虽然窗口滚动条有固定的宽度， Windows 使用 CreateWindow 调用中（或者在后面的 MoveWindow 调用中）给出的矩形尺寸来确定滚动条控制的尺寸。你可以建立细而长的滚动条控制，也可以建立短而粗的滚动条控制。

如果你想建立与窗口滚动条尺寸相同的滚动条控制，那么可以使用 GetSystemMetrics 获取水平滚动条的高度：

```
GetSystemMetrics (SM_CYHSCROLL) ;
```

或者垂直滚动条的宽度：

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

（根据 Windows 文档，滚动条窗口风格标识符 SBS_LEFTALIGN 、 SBS_RIGHTALIGN 、

SBS_TOPALIGN 和 SBS_BOTTOMALIGN 给出滚动条的标准尺寸，但是这些风格只在对话框中对滚动条有效。）

对窗口滚动条，你可以使用同样的调用来建立滚动条控制的范围和位置：

```
SetScrollRange (hwndScroll, SB_CTL, iMin, iMax, bRedraw) ;
SetScrollPos (hwndScroll, SB_CTL, iPos, bRedraw) ;
SetScrollInfo (hwndScroll, SB_CTL, &si, bRedraw) ;
```

其区别在于：窗口滚动条将父窗口的句柄作为第一个参数，并且 SB_VERT 或者 SB_HORZ 作为第二个参数。

令人吃惊的是，名为 COLOR_SCROLLBAR 的系统颜色不再用于滚动条，两端的按钮和小方块的颜色由 COLOR_BTNFACE 、 COLOR_BTNHIGHTLIGHT 、 COLOR_BTNSHADOW 、COLOR_BTNTEXT（用于小箭头）、COLOR_BTNSHADOW 和 COLOR_BTNLIGHT 决定，两端按钮之间区域的颜色由 COLOR_BTNFACE 和 COLOR_BTNHIGHLIGHT 决定。

如果你捕获了 WM_CTLCOLOR 消息，那么可以在消息处理中返回画刷以取代该颜色。让我们试一下。

8.4.1 COLORS1 程序

为了了解滚动条和静态子窗口的一些用法——也为了深入了解颜色——我们将使用 COLORS1 程序，如图 8-5 所示。COLORS1 在将三种滚动条显示在客户区的左半边，并分别标以“ Red ”、“ Green ”和“ Blue ”。当你滚动滚动条时，客户区的右半边将变为三种原色混合而成的合成色，三种原色的数值显示在三个滚动条的下面。

COLORS1.MAK

```
#-----------------------
# COLORS1.MAK make file
#-----------------------

colors1.exe : colors1.obj
     $(LINKER) $(GUIFLAGS) -OUT:colors1.exe colors1.obj $(GUILIBS)

colors1.obj : colors1.c
     $(CC) $(CFLAGS) colors1.c
```

COLORS1.C

```
/*----------------------------------------
   COLORS1.C -- Colors Using Scroll Bars
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>

LRESULT CALLBACK WndProc     (HWND, UINT, WPARAM, LPARAM) ;
```

```
LRESULT CALLBACK ScrollProc (HWND, UINT, WPARAM, LPARAM) ;

WNDPROC fnOldScr[3] ;
HWND    hwndScrol[3], hwndLabel[3], hwndValue[3], hwndRect ;
int     color[3], iFocus ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "Colors1" ;
     static char *szColorLabel[] = { "Red", "Green", "Blue" } ;
     HWND         hwnd ;
     int          i ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = CreateSolidBrush (0L) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Color Scroll",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     hwndRect = CreateWindow ("static", NULL,
                              WS_CHILD | WS_VISIBLE | SS_WHITERECT,
                              0, 0, 0, 0,
                              hwnd, (HMENU) 9, hInstance, NULL) ;

     for (i = 0 ; i < 3 ; i++)
          {
          hwndScrol[i] = CreateWindow ("scrollbar", NULL,
                              WS_CHILD | WS_VISIBLE | WS_TABSTOP | SBS_VERT,
                              0, 0, 0, 0,
                              hwnd, (HMENU) i, hInstance, NULL) ;

          hwndLabel[i] = CreateWindow ("static", szColorLabel[i],
                              WS_CHILD | WS_VISIBLE | SS_CENTER,
                              0, 0, 0, 0,
                              hwnd, (HMENU) (i + 3), hInstance, NULL) ;
          hwndValue[i] = CreateWindow ("static", "0",
                              WS_CHILD | WS_VISIBLE | SS_CENTER,
```

```
                            0, 0, 0, 0,
                            hwnd, (HMENU) (i + 6), hInstance, NULL) ;

          fnOldScr[i] = (WNDPROC) SetWindowLong (hwndScrol[i], GWL_WNDPROC,
                                                 (LONG) ScrollProc) ;

          SetScrollRange (hwndScrol[i], SB_CTL, 0, 255, FALSE) ;
          SetScrollPos   (hwndScrol[i], SB_CTL, 0, FALSE) ;
          }

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage  (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static COLORREF crPrim[3] = { RGB (255, 0, 0), RGB (0, 255, 0),
                                   RGB (0, 0, 255) } ;
     static HBRUSH   hBrush[3], hBrushStatic ;
     static int      cyChar ;
     static RECT     rcColor ;
     char            szbuffer[10] ;
     int             i, cxClient, cyClient ;

     switch (iMsg)
          {
          case WM_CREATE :
               for (i = 0 ; i < 3 ; i++)
                    hBrush[i] = CreateSolidBrush (crPrim[i]) ;

               hBrushStatic = CreateSolidBrush (
                                   GetSysColor (COLOR_BTNHIGHLIGHT)) ;

               cyChar = HIWORD (GetDialogBaseUnits ()) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               SetRect (&rcColor, cxClient / 2, 0, cxClient, cyClient) ;

               MoveWindow (hwndRect, 0, 0, cxClient / 2, cyClient, TRUE) ;

               for (i = 0 ; i < 3 ; i++)
                    {
                    MoveWindow (hwndScrol[i],
                         (2 * i + 1) * cxClient / 14, 2 * cyChar,
```

```
                         cxClient / 14, cyClient - 4 * cyChar, TRUE) ;

                    MoveWindow (hwndLabel[i],
                         (4 * i + 1) * cxClient / 28, cyChar / 2,
                         cxClient / 7, cyChar, TRUE) ;

                    MoveWindow (hwndValue[i],
                         (4 * i + 1) * cxClient / 28,
                         cyClient - 3 * cyChar / 2,
                         cxClient / 7, cyChar, TRUE) ;
                    }
               SetFocus (hwnd) ;
               return 0 ;

          case WM_SETFOCUS :
               SetFocus (hwndScrol[iFocus]) ;
               return 0 ;

          case WM_VSCROLL :
               i = GetWindowLong ((HWND) lParam, GWL_ID) ;

               switch (LOWORD (wParam))
                    {
                    case SB_PAGEDOWN :
                         color[i] += 15 ;
                                                  // fall through
                    case SB_LINEDOWN :
                         color[i] = min (255, color[i] + 1) ;
                         break ;

                    case SB_PAGEUP :
                         color[i] -= 15 ;
                                                  // fall through
                    case SB_LINEUP :
                         color[i] = max (0, color[i] - 1) ;
                         break ;

                    case SB_TOP :
                         color[i] = 0 ;
                         break ;

                    case SB_BOTTOM :
                         color[i] = 255 ;
                         break ;

                    case SB_THUMBPOSITION :
                    case SB_THUMBTRACK :
                         color[i] = HIWORD (wParam) ;
                         break ;

                    default :
                         break ;
                    }
               SetScrollPos  (hwndScrol[i], SB_CTL, color[i], TRUE) ;
               SetWindowText (hwndValue[i], itoa (color[i], szbuffer, 10)) ;
```

```
               DeleteObject ((HBRUSH)
                    SetClassLong (hwnd, GCL_HBRBACKGROUND,
                         (LONG) CreateSolidBrush (
                                   RGB (color[0], color[1], color[2])))) ;

               InvalidateRect (hwnd, &rcColor, TRUE) ;
               return 0 ;

          case WM_CTLCOLORSCROLLBAR :
               i = GetWindowLong ((HWND) lParam, GWL_ID) ;

               return (LRESULT) hBrush[i] ;

          case WM_CTLCOLORSTATIC :
               i = GetWindowLong ((HWND) lParam, GWL_ID) ;

               if (i >= 3 && i <= 8)    // static text controls
                    {
                    SetTextColor ((HDC) wParam, crPrim[i % 3]) ;
                    SetBkColor ((HDC) wParam,
                                   GetSysColor (COLOR_BTNHIGHLIGHT));

                    return (LRESULT) hBrushStatic ;
                    }
               break ;

          case WM_SYSCOLORCHANGE :
               DeleteObject (hBrushStatic) ;

               hBrushStatic = CreateSolidBrush (
                                   GetSysColor (COLOR_BTNHIGHLIGHT)) ;
               return 0 ;

          case WM_DESTROY :
               DeleteObject ((HBRUSH)
                    SetClassLong (hwnd, GCL_HBRBACKGROUND,
                         (LONG) GetStockObject (WHITE_BRUSH))) ;

               for (i = 0 ; i < 3 ; DeleteObject (hBrush[i++])) ;

               DeleteObject (hBrushStatic) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

LRESULT CALLBACK ScrollProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     int i = GetWindowLong (hwnd, GWL_ID) ;

     switch (iMsg)
          {
```

```
        case WM_KEYDOWN :
             if (wParam == VK_TAB)
                  SetFocus (hwndScrol[(i +
                       (GetKeyState (VK_SHIFT) < 0 ? 2 : 1)) % 3]) ;
             break ;

        case WM_SETFOCUS :
             iFocus = i ;
             break ;
        }
   return CallWindowProc (fnOldScr[i], hwnd, iMsg, wParam, lParam) ;
   }
```

图 8-5 COLORS1程序

COLORS1 利用子窗口进行工作，该程序使用 10 个子窗口控制：3 个滚动条、6 个静态文本窗口和 1 个静态矩形框。COLORS1 捕获 WM_CTLCOLORSCROLLBAR 消息来给红、绿、蓝三个滚动条的内部着色，并捕获 WM_CTLCOLORSTATIC 消息来着色静态文本。

你可以使用鼠标或者键盘来滚动滚动条，从而利用 COLORS1 作为一种开发工具尝试颜色的使用，为你自己的 Windows 程序选择漂亮的颜色（或者，你可能更喜欢，难看的颜色）。COLORS1 的显示如图 8-6 所示，不幸的是，这些颜色在打印纸上被显示为不同深度的灰色。

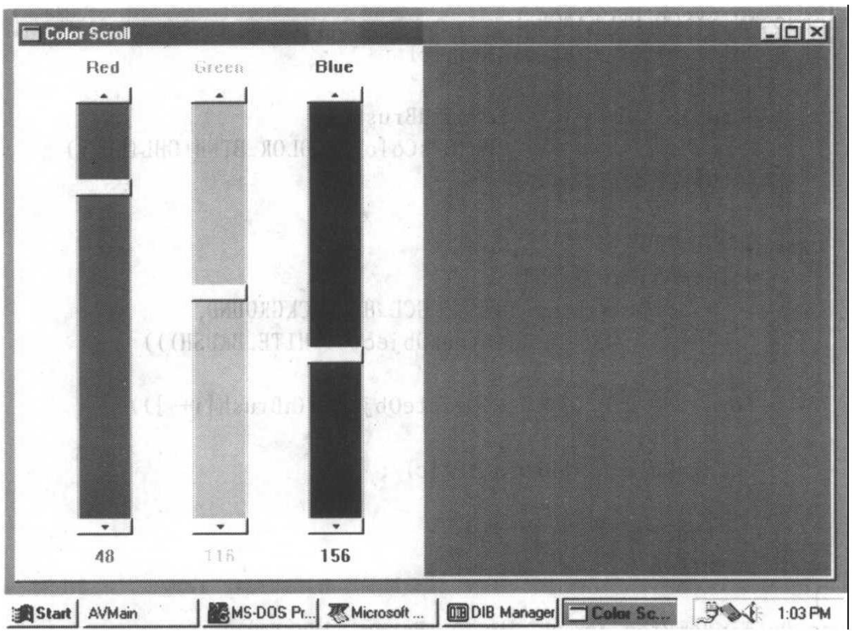


图 8-6 COLORS1显示

COLORS1 不处理 WM_PAINT 消息，它的几乎所有工作都是由子窗口完成的。

客户区右半边显示的颜色实际上是窗口的背景颜色。SS_WHITERECT 风格的静态子窗口显示在客户区的左半边。三个滚动条是 SBS_VERT 风格的子窗口控制，它们被定位在 SS_WHITERECT 子窗口的顶部。另外六个 SS_CENTER 风格（居中文本）的静态子窗口

提供标签和颜色值。COLORS1 在 WinMain 函数中用 CreateWindow 创建它的普通的重叠式窗口和 10 个子窗口。SS_WHITERECT 和 SS_CENTER 静态窗口使用窗口类 "static"；三个滚动条使用窗口类 "scrollbar"。

CreateWindow 调用中的 x 位置、y 位置、宽度和高度参数最初置为 0，因为位置和大小都取决于客户区的尺寸，而它目前尚未确定。COLORS1 的窗口过程在接收到 WM_SIZE 消息时，就使用 MoveWindow 给 10 个子窗口重新确定大小，所以，每当你对 COLORS1 窗口进行缩放时，滚动条的尺寸就会按比例变化。

当 WndProc 窗口过程收到 WM_VSCROLL 消息时，lParam 参数的高位字就是子窗口的句柄。我们可以使用 GetWindowLong 来得到子窗口的 ID 号：

```
I = GetWindowLong (lParam, GWW_ID) ;
```

对于这三个滚动条，我们已经按习惯将其 ID 置为 0、1、2，所以 WndProc 能区别出是哪个滚动条在产生消息。

由于子窗口的句柄在创建时就被保存在数组中，所以 WndProc 就能对相应滚动条消息进行处理，并通过调用 SetScrollPos 来设置相应的新值：

```
SetScrollPos (hwndScrol[i], SB_CTL, color[i], TRUE) ;
```

WndProc 也改变滚动条底部的子窗口的文本：

```
SetWindowText (hwndValue[i], itoa (color[i]), szbuffer, 10) ;
```

8.4.2 自动键盘接口

滚动条控制也能处理键盘输入，但是只能在拥有输入焦点时。下表说明怎样将键盘光标键转变为滚动条消息：

光标键	滚动条消息的 wParam 值
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
Page Down	SB_PAGEDOWN
左或上	SB_LINEUP
右或下	SB_LINEDOWN

事实上，SB_TOP 和 SB_BOTTOM 滚动条消息只能用键盘产生，在使用鼠标按动滚动条时，如果想使该滚动条获得输入焦点，那么你必须将 WS_TABSTOP 标识符包含到 CreateWindow 调用的窗口类参数中。当滚动条拥有输入焦点时，在该滚动条的小方框上将显示一个闪烁的灰色块。

为了给滚动条提供全面的键盘接口，还需要另外一些工作。首先，WndProc 窗口过程

必须使滚动条拥有输入焦点，它是通过处理 WM_SETFOCUS 消息来完成这一点的，该 WM_SETFOCUS 消息是当滚动条获得输入焦点时其父窗口接收到的。WindProc 给其中一个滚动条设置输入焦点。

```
SetFocus (hwndScrol[nFocus]) ;
```

但是，还需要一些借助键盘尤其是 Tab 键来从一个滚动条转换到另一个滚动条的方法。这比较困难，因为一旦某个滚动条拥有了输入焦点，它就处理所有的键盘输入，但滚动条只关心光标键，而忽略 Tab 键。解决这一两难处境的方法是“窗口子类化”。我们将用它来给 COLORS1 增加使用 Tab 键从一个滚动条跳到另一个滚动条的功能。

8.4.3 窗口子类化

滚动条控制的窗口过程是 Windows 内部的，但是，将 GWL_WNDPROC 标识符作为参数来调用 GetWindowLong ，你就可以得到这个窗口过程的地址。另外，你可以调用 SetWindowLong 给该滚动条设置一个新的窗口过程，这个技术叫做“窗口子类化”，非常有用。它能让你给现存的窗口过程设置“钩子”，以便在自己的程序中处理一些消息，同时将其它所有消息传递给旧的窗口过程。

在 COLORS1 中对滚动条消息进行预备处理的窗口过程叫做 ScrollProc ，它在 COLORS1.C 文件的尾部。由于 ScrollProc 是 COLORS1 中的函数，因此它必须被定义为回调函数。

对三个滚动条中的每一个， COLORS1 使用 SetWindowLong 来设置新的滚动条窗口过程的地址并获取现存滚动条窗口过程的地址：

```
fnoldScr[i] = (WNDPROC)SetWindowLong (hwndScroll[i].GWL_WNDPROC,
                                      (LONG)ScrollProc) ;
```

现在，函数 ScollProc 得到了 Windows 发送到 COLORS1 中三个滚动条（当然不是其它程序中的滚动条）的滚动条窗口过程的全部消息。 ScrollProc 窗口过程在接收到 Tab 或者 Shift-Tab 键时，就将输入焦点改变到下一个（或者上一个）滚动条。它使用 CallWindowProc 调用旧的滚动条窗口过程。

8.4.4 给背景着色

当 COLORS1 定义它的窗口类时，也为其客户区背景定义了一个实心的黑色画刷：

```
wndclass.hbrBackgroud = CreateSolidBrush (0L) ;
```

当你改变 COLORS1 的滚动条设置时，程序必须建立一个新的画刷，并将该新画刷句柄放入窗口类结构中。如同使用 GetWindowLong 和 SetWindowLong 能得到并设置滚动条窗口过程一样，用 GetClassWord 和 SetClassWord 能得到这个画刷的句柄。

你可以建立新的画刷并将其句柄插入窗口类结构中，然后删除旧的画刷：

```
DeleteObject ((HBRUSH)
    SetClassLong (hwnd, GCL_HBRBACKGROUND, (LONG)
        CreateSolidBrush (RGB (color[0], color[1], color[2])))) ;
```

Windows 下一次重新给窗口的背景着色时，将使用这个新画刷。为了强迫 Windows 抹掉背景，我们将使整个客户区无效：

```
InvalidateRect (hwnd, &rcColor, TRUE) ;
```

TRUE （非零）值作为第三个参数表示希望在重新着色之前删去背景。

InvalidateRect 使 Windows 在窗口过程的消息队列中放进一条 WM_PAINT 消息。由于 WM_PAINT 消息的级别比较低，所以，如果你还在使用鼠标或者光标键移动滚动条的话，这条消息将不会立即被处理。如果你想在颜色改变之后使该窗口立即变成最新的(当前的)，那么你可以在 InvalidateRect 之后增加下面的语句：

```
UpdateWindow (hwnd) ;
```

但是这会使得键盘和鼠标处理变慢。

COLORS1 中的 WndProc 函数不处理 WM_PAINT 消息，而是将其传给 DefWindowProc。Windows 对 WM_PAINT 消息的缺省处理只是调用 BeginPaint 和 EndPaint 使窗口生效。因为在 InvalidateRect 调用中已经指定背景要被抹掉，所以 BeginPaint 调用使 Windows 发出一条 WM_ERASEBKGND（删除背景）消息，WndProc 也将忽略这条消息。Windows 用窗口类中指定的画刷将客户区的背景抹去，这样就处理了这条消息。

在终止以前进行清除总是一个好主意，因此在处理 WM_DESTROY 消息期间，再一次调用 DeleteObject：

```
DeleteObject ((HBRUSH)
    SetClassLong (hwnd, GCL_HBRBACKGROUND,
        (LONG)GetStockObject (WHITE_BRUSH))) ;
```

8.4.5 给滚动条和静态文本着色

在 COLORS1 中，三个滚动条的内部和六个文本域中的文本着色为红、绿和蓝色。滚动条的着色是通过处理 WM_CTLCOLORSCROLLBAR 消息来完成的。

在 WndProc 中，我们为画刷定义了一个由三个句柄组成的静态数组：

```
static HBURSH hBrush[3];
```

在处理 WM_CREATE 期间，我们建立三个画刷：

```
for (I=0; I<3; I++)
    hBrush[i] = CreateSolidBrush (crPrim[i]) ;
```

其中 crPrim 数组中包含三种原色的 RGB 值。在 WM_CTLCOLORSCROLLBAR 处理期间，窗口过程返回这三个画刷中的一个：

```
case WM_CTLCOLORSCROLLBAR:
    i = GetWindowLong ((HWND)lParam, GWW_ID) ;
        return (LRESULT)hBrush[i];
```

在处理 WM_DESTROY 消息的过程中，这些画刷必须被删除：

```
for  (I=0; I<3; DeleteObject (hBrush[i++])) ;
```

同样，静态文本域中的文本是在处理 WM_CTLCOLORSTATIC 消息中调用 SetTextColor 来着色的。文本背景用 SetBkColor 函数设置为系统颜色 COLOR_BTNHIGHTLIGHT，这导致文本背景颜色和滚动条与文本后面的静态矩形控制的颜色一样。对于静态文本控制，这种文本背景颜色只用于字符串中每个字符后面的矩形，而不会用于整个控制窗口。为了实现这一点，窗口过程还必须返回 COLOR_BTNHIGHTLIGHT 颜色画刷的句柄。这个画刷被称为 hBrushStatic，它在 WM_CREATE 消息期间创建，在 WM_DESTROY 消息期间清除。

在 WM_CREATE 消息期间基于 COLOR_BTNHIGHTLIGHT 颜色创建画刷，并且在运行期间使用这一画刷时，我们遇到了一个小问题。如果程序在运行期间改变了 COLOR_BTNHIGHTLIGHT 颜色，那么静态矩形的颜色将发生变化，并且文本背景的颜色也会变化，但是文本窗口控制的整个背景将保持原有的 COLOR_BTNHIGHTLIGHT 颜色。

为了解决这个问题， COLORS1 简单地通过使用新颜色重新创建 hBrushStatic 也来处理 WM_SYSCOLORCHANGE 消息。

8.5 编 辑 类

在某些方面，编辑类是最简单的预定义窗口类；在另一方面，它又是最复杂的窗口类。当你使用类名“ edit ”建立子窗口时，你根据 CreateWindow 调用中的 x 位置、 y 位置、宽度和高度这些参数定义了一个矩形。当子窗口控制拥有输入焦点时，你可以打印文本，移动光标，使用鼠标或者 Shift 键与一个光标键来选取部分文本，按 Ctrl-X 来删除所选文本或按 Ctrl-C 来拷贝所选文本、并送到剪贴板上，按 Ctrl-V 键插入剪贴板上的文本。

编辑控制的最简单应用之一是作为单行输入区域。但是编辑控制并不仅限于单行，这一点我将在图 8-7 的 POPPAD1 程序中说明。和我们在这本书中所遇到的各种其它问题一样， POPPAD 程序将逐步增强来使用菜单、对话框（装入和保存文件）和打印机，最后的版本将是一个简单而完整的文本编辑器，并且其代码将非常简洁。

POPPAD1.MAK

```
#-----------------------
# POPPAD1.MAK make file
#-----------------------
```

```
poppad1.exe : poppad1.obj
     $(LINKER) $(GUIFLAGS) -OUT:poppad1.exe poppad1.obj $(GUILIBS)

poppad1.obj : poppad1.c
     $(CC) $(CFLAGS) poppad1.c
```

POPPAD1.C

```
/*-------------------------------------------------------
   POPPAD1.C -- Popup Editor using child window edit box
                (c) Charles Petzold, 1996
  -------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

char szAppName[] = "PopPad1" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, szAppName,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
```

```
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndEdit ;

     switch (iMsg)
          {
          case WM_CREATE :
               hwndEdit = CreateWindow ("edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                              WS_BORDER | ES_LEFT | ES_MULTILINE |
                              ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                         0, 0, 0, 0,
                         hwnd, (HMENU) 1,
                         ((LPCREATESTRUCT) lParam) -> hInstance, NULL) ;
               return 0 ;

          case WM_SETFOCUS :
               SetFocus (hwndEdit) ;
               return 0 ;

          case WM_SIZE :
               MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                                           HIWORD (lParam), TRUE) ;
               return 0 ;

          case WM_COMMAND :
               if (LOWORD (wParam) == 1)
                    if (HIWORD (wParam) == EN_ERRSPACE ||
                        HIWORD (wParam) == EN_MAXTEXT)
                              MessageBox (hwnd,
                                          "Edit control out of space.",
                                          szAppName, MB_OK | MB_ICONSTOP) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 8-7 POPPAD1程序

POPPAD1 是一个多行编辑器（只是没有文件输入/输出），其源代码不到 100 行。（不过，有一个缺陷，即预定义的多行编辑控制只限于 32KB 的文本。）你可以看到，POPPAD1 本身并没有做多少工作，预定义的编辑控制完成了许多工作，这样，你可以知道无需额外的程序时编辑控制能做些什么。

8.5.1 编辑类风格

如前面所提到的，在 CreateWindow 调用中将 " edit "作为窗口类建立了一个编辑控制，窗口风格是 WS_CHILD 加上几个选项。如同在静态子窗口控制中一样，编辑控制中的文本可以按左对齐、右对齐或者居中，你使用窗口风格 ES_LEFT 、 ES_RIGHT 和 ES_CENTER 来指定这种格式。

缺省地，编辑控制是单行的。你使用 ES_MULTILINE 窗口风格可以建立多行编辑控制。对于单行编辑控制，你一般只可以在编辑控制矩形的尾部输入正文。要建立一个自动水平滚动的编辑控制，你可以采用风格 ES_AUTOHSCROLL 。对 一个多行编辑控制，文本会整字回行，除非使用 ES_AUTOHSCROLL 风格，在这种情况下，你必须按 Enter 键来开始新的一行。你也可以试着用一下 ES_AUTOHSCROLL 风格，还可以使用风格 ES_AUTOVSCROLL 来将垂直滚动条包括在多行编辑控制中。

当你在多行编辑控制中包括这些滚动风格后，也许还想给编辑控制增加滚动条，要做到这些，可以对非子窗口使用同一窗口风格标识符 WS_HSCROLL 和 WS_VSCROLL 。

缺省地，编辑控制没有边界，使用风格 WS_BORDER 可以增加边界。

当你在编辑控制中选择文本时， Windows 将选择的文本反显。但是当编辑控制失去输入焦点时，被选择的文本将不再被加亮。如果你希望在编辑控制没有输入焦点时被选择的文本仍然被加亮，你可以使用风格 ES_NOHIDESEL 。

在 POPPAD1 创建其编辑控制时， CreateWindow 调用按如下形式给出风格：

```
WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
    WS_BORDER | ES_LEFT | ES_MULTILINE |
    ES_AUTOHSCROLL | ES_AUTOVSCROLL
```

在 POPPAD1 中，编辑控制的大小是后来当 WndProc 接收到 WM_SIZE 选项时通过调用 MoveWindow 来定义的。编辑控制的尺寸被简单地设置为主窗口的尺寸：

```
MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                            HIWORD (lParam), TRUE) ;
```

对于单行编辑控制，控制的高度必须可以容纳一个字符。如果编辑控制有边界（大多数都有），那么使用一个字符高度的 1.5 倍（包括外部间距(external leading)）。

8.5.2 编辑控制通知

编辑控制给父窗口过程发送 WM_COMMAND 消息，对按钮控制来说， wParam 和 lParam 变量的含义是相同的：

参数	描述
LOWORD (wParam)	子窗口 ID
HIWORD (wParam)	通知码
lParam	子窗口句柄

通知码如下所示:

EN_SETFOCUS	编辑控制已经获得输入焦点
EN_KILLFOCUS	编辑控制已经失去输入焦点
EN_CHANGE	编辑控制的内容将改变
EN_UPDATE	编辑控制的内容已经改变
EN_ERRSPACE	编辑控制运行已经超出空间
EN_MAXTEXT	编辑控制在插入时运行超出空间
EN_HSCROLL	编辑控制的水平滚动条已经被按动
EN_VSCROLL	编辑控制的垂直滚动条已经被按动

POPPAD1 只捕获 EN_ERRSPACE 通知码并显示一个消息框。

编辑控制将文本保存在父窗口过程的局部堆中，我上面已经提到过，编辑控制的内容限制在 32KB 之内。

8.5.3 使用编辑控制

如果在你的主窗口表面使用了几个单行编辑控制，那么你需要将窗口子类化以便把输入焦点从一个控制转移到另一个控制。你可以通过捕获 Tab 键和 Shift-Tab 键来完成这种移动，非常像 COLORS1 中所做的（另一个窗口子类化的例子在后面的 HEAD 程序中说明）。如何处理 Enter 键取决于你，可以像 Tab 键那样使用，也可以用作给程序的信号，表示所有的编辑区都准备好了。

如果你想在编辑区中插入文本，那么可以使用 SetWindowText 来做到。将文本从编辑控制中取出涉及到 GetWindowTextLength 和 GetWindowText，我们将在 POPPAD 程序的修订版本中看到这些操作的实例。

8.5.4 发送给编辑控制的消息

因为用 SendMessage 发送给编辑控制的消息很多，并且其中的几个还将在后面的 POPPAD 修订版本中用到，所以这里不涉及所有用 SendMessage 发送给编辑控制的消息，只概要地说明一下。

这些消息允许你剪取、拷贝或者清除当前选择。用户使用鼠标或者 Shift 键加上光标控制键来选择文本并进行上面的操作，这样，在编辑控制中选中的文本将被加亮:

```
SendMessage (hwndEdit, WM_CUT, 0, 0) ;
SendMessage (hwndEdit, WM_COPY, 0, 0) ;
SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
```

WM_CUT 将当前选择从编辑控制中移走，并将其发送到剪贴板中; WM_COPY 将选择的文本拷贝到剪贴板上并保持编辑控制中的内容完好无损; WM_CLEAR 将选择的内容从编

辑控制中删除，但是不向剪贴板中发送。

你也可以将剪贴板上的文本插入到编辑控制中的光标位置：

```
SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
```

你可以获取当前选择的起始位置和末尾位置：

```
SendMessage (hwndEdit, EM_GETSEL, (WPARAM)&iStart,
                                   (LPARAM)&iEnd) ;
```

结束位置实际上是最后一个选择字符的位置加 1 。

你可以选择文本：

```
SendMessage (hwndEdit, EM_SETSEL, iStart, iEnd) ;
```

你还可以使用别的文本来置换当前的选择：

```
SendMessge (hwndEdit, EM_REPLACESEL, 0, (LPARAM)szString) ;
```

对多行编辑控制，你可以获取行数：

```
iCount = SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0) ;
```

对任何特定的行，你可以获取距离编辑缓冲区文本开头的偏移量：

```
iOffset = SendMessage (hwndEdit, EM_LINEINDEX, iLine, 0) ;
```

行数从 0 开始计算， iLine 值为-1 时返回包含光标的行的偏移量。你可以获取行的长度：

```
iLength = SendMessage (hwndEdit, EM_LINLENGTH, iLine, 0) ;
```

并将行本身拷贝到一个缓冲区中：

```
iLength = SendMessage (hwndEdit, EM_GETLINE, iLine,
                       (LPARAM) szBuffer) ;
```

8.6 列表框类

我在本章讨论的最后一个预定义子窗口控制是列表框。一个列表框是文本串的集合，这些文本串是一个矩形中可以滚动显示的列状列表。一程序通过向列表框窗口过程发送消息，可以在列表中增加或者删除串。当列表框中的某项被选择时，列表框控制就向其父窗口发送 WM_COMMAND 消息，父窗口也就可以确定选择的是哪一项。

列表框最常用于对话框中，例如，从 File 菜单中选择 Open 即可以调出对话框。列表框能显示当前目录中的文件，也能显示其它子目录。一个列表框可以是单选的，也可以是多选的，后者允许用户从列表框中选择多个项。当列表框拥有输入焦点时，其中的项的周围显示有虚线。在列表框中，光标位置并不指明被选择的项。被选择的项被加亮显示，并且是反显的。

在单项选择的列表框中，用户按空格键就可以选择光标所在位置的项。方向键移动光标和当前选择指示，并且能够滚动列表框的内容。PageUp 和 PageDown 键也能滚动列表框，但它移动的是光标而不是选择指示。按字母键能将光标和选择指示移到以此字母开头的第

一个（或下一个）选项。也可以使用鼠标在要选择的项上点一下或者两下来选择它。

在多项选择列表框中，空格键可以切换光标所在位置的项的选择状态（如果该项已经被选择，则取消选择）。如同在单项选择列表框中一样，方向键取消前面选择过的项，并且移动光标和选择指示。但是，Ctrl 键和方向键能够在移动光标的同时不移动选择，Shift 键加方向键能扩展一个选择。

在多项选择列表框中，单击或者双击鼠标键能取消以前所有的选择，而选择被点中的项。但是，如果在鼠标点中某一项的同时也按下 Shift 键，则只能切换该项的选择状态，而不会改变任何其它项的选择状态。

8.6.1 列表框风格

当你使用 CreateWindow 建立列表框子窗口时，你应该将"listbox"作为窗口类，将 WS_CHILD 作为窗口风格。但是，这个缺省列表框风格不向其父窗口发送 WM_COMMAND 消息，这样一来，程序必须向列表框询问其中的项的选择状态（借助于发送给列表框控制的消息）。所以，列表框控制通常都包括列表框风格标识符 LBS_NOTIFY，它允许父窗口接收来自列表框的 WM_COMMAND 消息。如果你希望列表框控制对列表框中的项进行排序，那么你可以使用另一种常用的风格 LBS_SORT。

缺省地，列表框是单项选择的。多项选择的列表框相当少。如果你想建立一个多项选择列表框，那么你可以使用风格 LBS_MULTIPLESEL。

通常，当给滚动框增加新项时，列表框会修改它自身。你可以通过将风格 LBS_NOREDRAW 包含进去来防止这种现象。但是你也许不想使用这种风格，这时可以使用 WM_SETREDRAW 消息来暂时防止列表框控制的重新着色，我将在稍后讨论 WM_SETREDRAW 消息。

缺省地，列表框窗口过程只显示列表项，它的周围没有任何边界。你可以使用窗口风格标识符 WS_BORDER 来加上边界。另外，你可以使用窗口风格标识符 WS_VSCROLL 来增加垂直滚动条，以便用鼠标来滚动列表项。

Windows 头文件定义了一个列表框风格，叫做 LBS_STANDARD，它包含了最常用的风格，其定义如下：

```
(LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER)
```

你也可以采用 WS_SIZEBOX 和 WS_CAPTIOIN 标识符，但是这两个标识符允许你重新定义列表框的大小，也允许你在列表框父窗口的客户区中移动列表框。

列表框的宽度应该能容纳最长串的宽度加上滚动条的宽度。你可以使用：

```
GetSystemMetrics (SM_CXVSCROLL) ;
```

来获得垂直滚动条的宽度。你用一个字符的高度乘以想要在视口中显示的项数来计算出列表框的高度。

8.6.2　将串放入列表框

创建列表框之后，下一步是将文本串放入其中，你可以通过调用 SendMessage 给列表框窗口过程发消息来做到这一点。文本串通常以 0 开始计数的下标数来引用，其中 0 对应于最顶上的项。在下面的例子中， hwndList 是子窗口列表框控制的句柄，而 iIndex 是下标值。在使用 SendMessage 传递文本串的情况下， lParam 参数是指向 0 结尾串的指针。

在大多数例子中，当窗口过程存储的列表框内容超过了可用内存空间时，SendMessage 将返回 LB_ERRSPACE （定义为－2）。如果是因为其它原因出的错，那么 SendMessage 将返回 LB_ERR（－1）。如果操作成功，那么 SendMessage 将返回 LB_OKAY（0）。你可以通过测试 SendMessage 的非零值来判明这两种错误。

如果你采用 LBS_SORT 风格（或者如果你在列表框中按照想要呈现的顺序排列字符串），那么填充列表框最简单的方法是借助 LB_ADDSTRING 消息：

```
SendMessage (hwndList, LB_ADDSTRING, 0,  (LPARAM)szString) ;
```

如果你没有采用 LBS_SORT ，那么可以使用 LB_INSERTSTRING 指定一个下标值，将字符串插入到列表框中：

```
SendMessage (hwndList, LB_INSERTSTRING, iIndex,  (LPARAM)szString) ;
```

例如，如果 iIndex 等于 4 ，那么 szString 将变为下标值为 4 的串——从顶头开始算起的第五个串（因为是从 0 开始计数的），位于这个点后面的所有串都将向后推移。下标值为-1时，将串增加在最后。你可以对风格为 LBS_SORT 的列表框使用 LB_INSERTSTRING ，但是这个列表框的内容不能被重新排序。（你也可以使用 LB_DIR 消息将串插入到列表框中，这将在本章的最后进行讨论。）

你可以在指定下标值的同时使用 LB_DELETESTRING 参数，这就可以从列表框中删除串：

```
SendMessage (hwndList, LB_DELETESTRING, iIndex, 0) ;
```

你可以使用 LB_RESETCONTENT 清除列表框中的内容：

```
SendMessage (hwndList, LB_RESETCONTENT, 0, 0L) ;
```

当在列表框中增加或者删除串时，列表框窗口过程将更新显示。如果你有许多串需要增加或者删除，那么你也许希望暂时阻止这一动作，其方法是关掉控制的重画标志：

```
SendMessage (hwndList, WM_SETREDRAW, FALSE, 0) ;
```

当你完成后，可以再打开重画标志：

```
SendMessage (hwndList, WM_SETREDRAW, TRUE, 0) ;
```

使用 LBS_NOREDRAW 风格建立的列表框开始时其重画标志是关闭的。

8.6.3 选择和获取项

SendMessage 完成了下面所描述的任务之后，通常返回一个值。如果出错，那么这个值将被设置为 LB_ERR （定义为-1 ）。

当列表框中放入一些项之后，你可以弄清楚列表框中有多少项：

```
iCount = SendMessage (hwndList, LB_GETCOUNT, 0, 0) ;
```

其它一些调用对单项选择列表框和多项选择列表框是不同的。让我们先来看看单项选择列表框。

通常，你让用户在列表框中选择条目。但是如果你想加亮显示一个缺省选择，则可以使用：

```
SendMessage (hwndList, LB_SETCURSEL, iIndex, 0) ;
```

将 iIndex 设置为-1 则取消所有选择。

你也可以根据项的第一个字母来选择：

```
iIndex = SendMessage (hwndList, LB_SELECTSTRING, iIndex,
                     (LPARAM)szSearchString) ;
```

在 SendMessage 调用中将 iIndex 作为 lParam 参数时， iIndex 是下标，可以根据它搜索其开头字母与 szSearchString 相匹配的项。 iIndex 的值等于-1 时从头开始搜索， SendMessage 返回被选中项的下标。如果没有开头字母与 szSearchString 相匹配的项时， SendMessage 返回 LB_ERR 。

当你得到来自列表框的 WM_COMMAND 消息时（或者在任何其它时候），你可以使用 LB_GETCURSEL 来确定当前选项的下标：

```
iIndex = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
```

如果没有项被选中，那么从调用中返回的 iIndex 值为 LB_ERR 。

你可以确定列表框中串的长度：

```
iLength = SendMessage (hwndList, LB_GETTEXTLEN, iIndex, 0) ;
```

并可以将某项拷贝到文本缓冲区中：

```
iLength = SendMessage (hwndList, LB_GETTEXT, iIndex,
                      (LPARAM)szBuffer) ;
```

在这两种情况下，从调用返回的 iLength 值是字符串的长度。对以 NULL 终结的串的长度

来说， szBuffer 数组必须足够大。你也许想用 LB_GETTEXTLEN 先分配一些局部内存来存放串。

对一个多项选择列表框，你不能使用 LB_SETCURSEL 、 LB_GETCURSEL 或者 LB_SELECTSTRING，但是你可以使用 LB_SETSEL 来设置某特定项的选择状态，而不影响有可能被选择的其它项：

```
SendMessage (hwndList, LB_SETSEL, wParam, iIndex) ;
```

wParam 参数不为零时，选择并加亮某一项； wParam 为 0 时，取消选择。如果 lParam 等于 -1，那么将选择所有项或者取消所有被选中的项。你可以如下确定某特定项的选择状态：

```
iSelect = SendMessage (hwndList, LB_GETSEL, iIndex, 0) ;
```

其中，如果由 iIndex 指定的项被选中， iSelect 被置为非 0，否则被置为 0。

8.6.4 接收来自列表框的消息

当用户用鼠标单击列表框时，列表框将接受输入焦点。下面的操作可以使父窗口将输入焦点转交给列表框控制：

```
SetFocue (hwndList) ;
```

当列表框拥有输入焦点时，光标移动键、字母键和空格键都可以用来在该列表框中选择某项。

列表框控制向其父窗口发送 WM_COMMAND 消息，对按钮和编辑控制来说， wParam 和 lParam 变量的含义是相同的：

LOWORD (*wParam*)	子窗口 ID
HIWORD (*wParam*)	通知码
lParam	子窗口句柄

通知码及其值如下所示：

LBN_ERRSPACE	-2
LBN_SELCHANGE	1
LBN_DBLCLK	2
LBN_SELCANCEL	3
LBN_SETFOCUS	4
LBN_KILLFOCUS	5

只有列表框窗口风格包括 LBS_NOTIFY 时，列表框控制才向父窗口发送 LBN_SELCHANGE 和 LBN_DBLCLK 码。

LBN_ERRSPACE 码表明列表框已经超出运行空间。LBN_SELCHANGE 码表明当前选择已经被改变；这些消息出现在下列情况下：用户在列表框中移动加亮的项时，用户使用空格键切换选择状态时或者用鼠标单击某项时。LBN_DBLCLK 码说明某项已经被鼠标双击。（LBN_SELCHANGE 和 LBN_DBLCLK 通知码的值表示鼠标按动的次数。）

根据应用的需要，你也许要使用 LBN_SELCHANGE 或 LBN_DBLCLK，也许二者都要使用。你的程序会收到许多 LBN_SELCHANGE 消息，但是 LBN_DBLCLK 消息只有当用户双击鼠标时才会出现。如果你的程序使用双击，那么你需要提供一个复制 LBN_DBLCLK 的键盘接口。

8.6.5 一个简单的列表框应用程序

既然你知道了如何建立列表框，如何使用文本项填充列表框，如何接收来自列表框的控制以及如何获取串，现在是到了写一个应用程序的时候了。如图 8-8 中所示，ENVIRON 程序在客户区中使用列表框来显示当前 MS-DOS 环境变量的名字（诸如 PATH 、COMSPEC 和 PROMPT 等）。当你选择一个变量时，名字和环境串被显示在客户区的顶部。

ENVIRON.MAK

```
#-----------------------
# ENVIRON.MAK make file
#-----------------------

environ.exe : environ.obj
     $(LINKER) $(GUIFLAGS) -OUT:environ.exe environ.obj $(GUILIBS)

environ.obj : environ.c
     $(CC) $(CFLAGS) environ.c
```

ENVIRON.C

```
/*----------------------------------------
   ENVIRON.C -- Environment List Box
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>

#define  MAXENV  4096

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Environ" ;
     HWND        hwnd ;
```

```
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Environment List Box",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szBuffer[MAXENV + 1] ;
     static HWND hwndList, hwndText ;
     HDC         hdc ;
     int         i ;
     TEXTMETRIC  tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               ReleaseDC (hwnd, hdc) ;

               hwndList = CreateWindow ("listbox", NULL,
                              WS_CHILD | WS_VISIBLE | LBS_STANDARD,
                              tm.tmAveCharWidth, tm.tmHeight * 3,
                              tm.tmAveCharWidth * 16 +
                                   GetSystemMetrics (SM_CXVSCROLL),
```

```
                             tm.tmHeight * 5,
                             hwnd, (HMENU) 1,
                             (HINSTANCE) GetWindowLong (hwnd,
                                                        GWL_HINSTANCE),
                             NULL) ;

               hwndText = CreateWindow ("static", NULL,
                             WS_CHILD | WS_VISIBLE | SS_LEFT,
                             tm.tmAveCharWidth,          tm.tmHeight,
                             tm.tmAveCharWidth * MAXENV, tm.tmHeight,
                             hwnd, (HMENU) 2,
                             (HINSTANCE) GetWindowLong (hwnd,
                                                        GWL_HINSTANCE),
                             NULL) ;

               for (i = 0 ; environ[i] ; i++)
                    {
                    if (strlen (environ[i]) > MAXENV)
                         continue ;
                    *strchr (strcpy (szBuffer, environ [i]), '=') = '\0' ;
                    SendMessage (hwndList, LB_ADDSTRING, 0,
                                 (LPARAM) szBuffer) ;
                    }
               return 0 ;

          case WM_SETFOCUS :
               SetFocus (hwndList) ;
               return 0 ;

          case WM_COMMAND :
               if (LOWORD (wParam) == 1 && HIWORD (wParam) == LBN_SELCHANGE)
                    {
                    i = SendMessage (hwndList, LB_GETCURSEL, 0, 0) ;
                    i = SendMessage (hwndList, LB_GETTEXT, i,
                                          (LPARAM) szBuffer) ;

                    strcpy (szBuffer + i + 1, getenv (szBuffer)) ;
                    *(szBuffer + i) = '=' ;

                    SetWindowText (hwndText, szBuffer) ;
                    }
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 8-8 ENVIRON程序

ENVIRON 创建两个子窗口：一个是 LBN_STANDARD 风格的列表框，另一个是 SS_LEFT 风格（左对齐文本）的静态窗口。ENVIRON 使用 environ 变量（在 STDLIB.H 中被申明为外部变量）来获得环境串的列表，并使用 LB_ADDSTRING 消息来指示列表框

窗口过程将每个串放入列表框中。

当你运行 ENVIRON 时，可以使用鼠标或者键盘来选择环境变量。每次你改变选择时，列表框都给其父窗口 WndProc 发送一条 WM_COMMAND 消息。当 WndProc 收到 WM_COMMAND 消息时，它就检查 wParam 是否为 1（列表框的子窗口 ID）和 lParam 的高位字（通知码）是否等于 LBN_SELCHANGE 。如果是的，那么它就使用 LB_GETCURSEL 消息来获得选中项目的下标，并使用 LB_GETTEXT 来获得外部环境变量名的文本串本身。ENVIRON 程序使用 C 语言函数 getenv 来获得与变量相对应的环境串，使用 SetWindowText 将该串传递到静态子窗口控制中，这个静态子窗口控制被用来显示文本。

注意， ENVIRON 不能用 LB_GETCURSEL 返回的下标值来检索 environ 变量和获取环境串。因为该列表框的风格为 LBS_SORT （包含在 LBS_STANDARD 内），所以两种下标不再匹配。

8.6.6 文件列表

我将最好的留在最后： LB_DIR ，这是功能最强的列表框消息。它用文件目录表填充列表框，并且可以选择将子目录和有效的磁盘驱动器也包括进来：

```
SendMessage (hwndList, LB_DIR, iAttr, (LPARAM)szFileSpec) ;
```

8.6.6.1 使用文件属性码

iAttr 参数是文件属性代码，其最低字节是进行 MS-DOS 函数调用时用到的普通的文件属性代码：

iAttr	值	属性
DDL_READWRITE	0x0000	普通文件
DDL_READONLY	0x0001	只读文件
DDL_HIDDEN	0x0002	隐含文件
DDL_SYSTEM	0x0003	系统文件
DDL_DIRECTORY	0x0004	子目录文件
DDL_ARCHIVE	0x0005	有归档位集的文件

高字节提供了一些对所要求的项的附加控制：

iAttr	值	选择
DDL_DRIVE	0x4000	包括驱动器的字母
DDL_EXCLUSIVE	0x8000	互斥搜索

当 LB_DIR 消息的 iAttr 值为 DDL_READWRITE 时，列表框列出普通文件、只读文件

和归档位置位的文件，这与利用 MS-DOS 函数调用来寻找文件的方法一致。当值为 DDL_DIRECTORY 时，列表框除列出上述文件之外，还列出子目录，目录位于方括号之内。当值为 DDL_DRIVE | DDL_DIRECTORY 时，那么列表将扩展到包括所有有效的驱动器，驱动器字母显示在虚线之间。

将 iAttr 的最高位置位就可以列出带标志的文件，而不包括普通文件。例如，对 Windows 的文件备份程序，也许你只想列出最后一次备份后修改过的文件，这种文件的归档位置位，因此你可以使用 DDL_EXCLUSIVE | DDL_ARCHIVE 。

8.6.6.2 文件列表的排序

lParam 参数是指向文件说明串如“*.*”的指针，这个文件说明串不影响列表框中的子目录。

你也许希望给列有文件清单的列表框使用 LBS_SORT 消息。列表框首先列出符合文件说明的文件，再（可选）使用下面的形式列出磁盘驱动器：

```
[-A-]
```

也可以选择列出子目录名，第一个列出的子目录为如下形式：

```
[..]
```

这一“两个点”的子目录项允许用户向根目录返回一级。（在根目录下列出文件名时这一“两个点”的项不会出现。）最后，具体的子目录名采用下面的形式：

```
[SUBDIR]
```

如果你不使用 LBS_SORT ，那么文件名和子目录名将被混在一起，而驱动器字母将出现在列表框的底部。

8.6.7 Windows 的 Head 程序

UNIX 中有一个著名的实用程序叫做 head ，它显示文件的开始几行。让我们使用列表框为 Windows 编写一个类似的程序。如图 8-9 中所示， HEAD 将所有文件和子目录列在列表框中。你可以挑选某个被选择的文件来显示，方法是在该文件上使用鼠标双击或者使用 Enter 键按动要选文件；你也可以使用这两种方法中的任何一种来改变子目录。这个程序在 HEAD 窗口客户区的右边从文件的开头开始显示，最多能够显示 8KB 的内容。

HEAD.MAK

```
#--------------------
# HEAD.MAK make file
#--------------------

head.exe : head.obj
     $(LINKER) $(GUIFLAGS) -OUT:head.exe head.obj $(GUILIBS)
```

```
head.obj : head.c
     $(CC) $(CFLAGS) head.c
```

HEAD.C

```
/*-----------------------------------------------
   HEAD.C -- Displays beginning (head) of file
             (c) Charles Petzold, 1996
  -----------------------------------------------*/

#include <windows.h>
#include <string.h>
#include <direct.h>
#define MAXPATH     256
#define MAXREAD    8192

LRESULT CALLBACK WndProc  (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ListProc (HWND, UINT, WPARAM, LPARAM) ;

WNDPROC fnOldList ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Head" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "File Head",
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
```

```
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL       bValidFile ;
     static char       sReadBuffer[MAXREAD], szFile[MAXPATH] ;
     static HWND       hwndList, hwndText ;
     static OFSTRUCT ofs ;
     static RECT       rect ;
     char              szBuffer[MAXPATH + 1] ;
     HDC               hdc ;
     int               iHandle, i ;
     PAINTSTRUCT       ps ;
     TEXTMETRIC        tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
               GetTextMetrics (hdc, &tm) ;
               ReleaseDC (hwnd, hdc) ;

               rect.left = 20 * tm.tmAveCharWidth ;
               rect.top  =  3 * tm.tmHeight ;

               hwndList = CreateWindow ("listbox", NULL,
                              WS_CHILDWINDOW | WS_VISIBLE | LBS_STANDARD,
                              tm.tmAveCharWidth, tm.tmHeight * 3,
                              tm.tmAveCharWidth * 13 +
                                   GetSystemMetrics (SM_CXVSCROLL),
                              tm.tmHeight * 10,
                              hwnd, (HMENU) 1,
                              (HINSTANCE) GetWindowLong (hwnd,
                                                   GWL_HINSTANCE),
                              NULL) ;

               hwndText = CreateWindow ("static",
                              getcwd (szBuffer, MAXPATH),
                              WS_CHILDWINDOW | WS_VISIBLE | SS_LEFT,
                              tm.tmAveCharWidth,           tm.tmHeight,
                              tm.tmAveCharWidth * MAXPATH, tm.tmHeight,
                              hwnd, (HMENU) 2,
                              (HINSTANCE) GetWindowLong (hwnd,
                                                   GWL_HINSTANCE),
                              NULL) ;

               fnOldList = (WNDPROC) SetWindowLong (hwndList, GWL_WNDPROC,
                                             (LPARAM) ListProc) ;

               SendMessage (hwndList, LB_DIR, 0x37, (LPARAM) "*.*") ;
```

```
          return 0 ;

     case WM_SIZE :
          rect.right  = LOWORD (lParam) ;
          rect.bottom = HIWORD (lParam) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hwndList) ;
          return 0 ;

     case WM_COMMAND :
          if (LOWORD (wParam) == 1 && HIWORD (wParam) == LBN_DBLCLK)
               {
               if (LB_ERR == (i = SendMessage (hwndList,
                                        LB_GETCURSEL, 0, 0L)))
                    break ;

               SendMessage (hwndList, LB_GETTEXT, i,
                            (LPARAM) szBuffer) ;

               if (-1 != OpenFile (szBuffer, &ofs, OF_EXIST | OF_READ))
                    {
                    bValidFile = TRUE ;
                    strcpy (szFile, szBuffer) ;
                    getcwd (szBuffer, MAXPATH) ;
                    if (szBuffer [strlen (szBuffer) - 1] != '\\')
                         strcat (szBuffer, "\\") ;
                    SetWindowText (hwndText,
                                   strcat (szBuffer, szFile)) ;
                    }
               else
                    {
                    bValidFile = FALSE ;
                    szBuffer [strlen (szBuffer) - 1] = '\0' ;
                    chdir (szBuffer + 1) ;
                    getcwd (szBuffer, MAXPATH) ;
                    SetWindowText (hwndText, szBuffer) ;
                    SendMessage (hwndList, LB_RESETCONTENT, 0, 0L) ;
                    SendMessage (hwndList, LB_DIR, 0x37,
                                 (LONG) "*.*") ;
                    }
               InvalidateRect (hwnd, NULL, TRUE) ;
               }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;
          SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
          SetTextColor (hdc, GetSysColor (COLOR_BTNTEXT)) ;
          SetBkColor   (hdc, GetSysColor (COLOR_BTNFACE)) ;

          if (bValidFile && -1 != (iHandle =
                    OpenFile (szFile, &ofs, OF_REOPEN | OF_READ)))
               {
```

```
                i = _lread (iHandle, sReadBuffer, MAXREAD) ;
                _lclose (iHandle) ;
                DrawText (hdc, sReadBuffer, i, &rect, DT_WORDBREAK |
                          DT_EXPANDTABS | DT_NOCLIP | DT_NOPREFIX) ;
                }
            else
                bValidFile = FALSE ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }

LRESULT CALLBACK ListProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
    if (iMsg == WM_KEYDOWN && wParam == VK_RETURN)

        SendMessage (GetParent (hwnd), WM_COMMAND, 1,
                     MAKELONG (hwnd, LBN_DBLCLK)) ;

    return CallWindowProc (fnOldList, hwnd, iMsg, wParam, lParam) ;
    }
```

图 8-9 HEAD程序

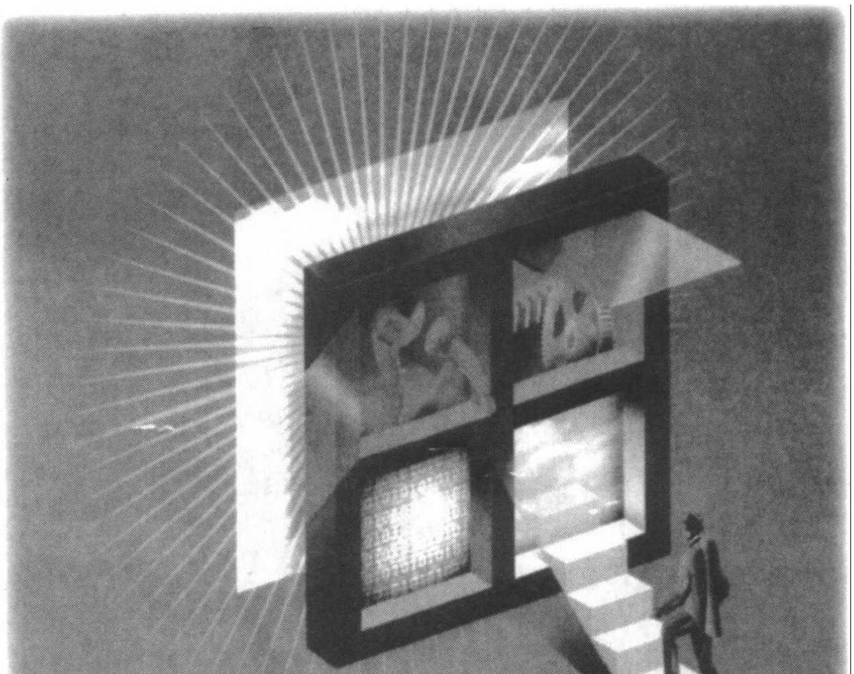
在 ENVIRON 中，当我们选择一个环境变量时——无论是用鼠标还是用键盘——程序都将显示一个环境串。但是，如果我们在 HEAD 中使用这种选择显示方法，那么程序响应会很慢，这是因为在列表框中移动选择时，程序仍然要不断地打开和关闭文件。然而，HEAD 要求文件或者子目录被双击，从而引起一些问题，这是因为列表框控制没有鼠标双击的自动键盘接口。前面讲过，如果可能，应该尽量提供键盘接口。

解决的方法是什么呢？当然是窗口子类化。HEAD 中的列表框子类函数叫做 ListProc，它寻找 wParam 参数等于 VK_RETURN 的 WM_KEYDOWN 消息，并给其父窗口发送一条带 LBN_DBLCLK 通知码的 WM_COMMAND 消息。在 WndProc 中，对 WM_COMMAND 的处理使用了 Windows 函数的 OpenFile 来检查列表框中的选择。如果 OpenFile 返回一个错误信息，则表明该选择不是文件，而可能是一个子目录。然后 HEAD 使用 chdir 来改变这个子目录。它向列表框发送一条 LB_RESETCONTENT 消息来清除其中的内容，再发送一条 LB_DIR 消息，使用新子目录中的文件来填充列表框。

WndProc 中的 WM_PAINT 消息是用 Windows 的 OpenFile 函数来打开文件的，这将返回一个 MS-DOS 文件句柄，该句柄可以传递给 Windows 函数_lread 和_lclose 。文件的内容用 DrawText 来显示。

第三部分

使用资源



第 9 章　图标、光标、位图和字符串

大多数基于 Windows 的程序都包含一个定制的图标，Windows 将该图标显示在应用程序窗口标题条的左上角。当程序被列在 Start 菜单中，或者被显示在屏幕底部的任务条中，或者被列在 Windows Explorer 中时，Windows 也显示该程序的图标。有些程序——诸如 Windows Paint 一类的重要的图形绘制工具——也使用用户化的鼠标光标来表示程序的不同操作。许许多多 Windows 程序还使用菜单和对话框。菜单、对话框加上滚动条，这是标准 Windows 用户界面的面包和黄油。

图标、光标、菜单和对话框都是相互关联的，它们是 Windows 的全部资源类型。资源即数据，它们被存储在程序的.EXE 文件中，但是它们并非驻留在程序的常规数据段中。也就是说，资源不能通过程序源代码中定义的变量直接访问到，它们必须明确地从.EXE 文件中装入内存。

当 Windows 把程序装入内存执行时，它通常将资源留在磁盘上。只有当 Windows 需要某一资源时，它才将资源装入内存。当然，你使用 Windows 程序时，或许已经注意到了资源的动态装入。当你第一次调用程序的对话框时，Windows 通常要读盘，以便从程序的.EXE 文件中把对话框资源拷贝到内存中。

在本书中，我们将讨论这些资源：

- 图标
- 光标
- 位图
- 字符串
- 用户定义资源
- 菜单
- 键盘加速键
- 对话框

上面所列的前五种资源将在本章中讨论，菜单和键盘加速键将在第 10 章介绍，而对话框将在第 11 章介绍。

9.1 编 译 资 源

在程序的开发过程中，资源是在一个“资源描述文件”中定义的。资源描述文件是以.RC 为扩展名的 ASCII 码文件，可以包含用 ASCII 码表示的资源，也可以引用其它资源描述文件（ASCII 或者二进制文件）。资源编译程序（RC.EXE）把资源文件编译成二进制形式，

这是以.RES 为扩展名的文件。通过在 LINK 命令行中指定.RES 文件，你可以使 LINK 将编译产生的资源和普通的程序代码，以及.OBJ 和.LIB 文件中的数据连接在一起，从而得到.EXE 文件。

在命令行上，你可以将.RC 资源文件编译成二进制的.RES 文件：

```
RC -r -DWIN32 filename.RC
```

该命令行创建一个名为 filename.RES 的二进制文件，大多数 Windows 程序都把资源文件命名为与程序本身相同的名字。

在上面所示的命令行中，－ r 参数使资源编译器创建.RES 文件并将它存储在磁盘上，虽然这是可选的，但你几乎总是这样做。－ DWIN32 参数定义一个常数（ WIN32 ），指示编译产生的资源应该适合 Windows 95 和 Windows NT 的 32 位文件格式存储。定义 WIN32 常数的一个含义是：在菜单和对话框中显示的文本串是以 Unicode 格式存储的，每个字符 2 字节。如果对一个.RES 文件进行十六进制的显示，那么你将看到菜单和对话框文本中所有的 ASCII 字符都是以 0 分隔的。不要惊讶——这是正常的。

在 Windows 95 之前，资源编译器还负责将编译后的资源加入连接程序所创建的.EXE 文件中。 Windows 95 版本的 Visual C++提供的连接程序完成整个工作，只要在 LINK 命令行中包含.OBJ 文件和.RES 文件名就可以了。

编译资源的过程是由我们目前熟悉的 make 文件来表示的。除了编译.C 源代码文件外，你还应该如下编译资源：

```
progname.res : progname.rc [progname.h] [和其它文件]
     $(RC) $(RCVARS) progname.rc
```

RC 和 RCVARS 宏由环境变量分别定义为“ rc ”和“-r-DWIN32 ”。这里，.H 头文件也可能是依赖文件。在资源文件中，你当然可以包含一个头文件，这个头文件通常也被包含在.C 源代码文件中，定义程序用来引用资源的所有标识符。依赖文件列表还可以包括“其它文件”，这些是在资源文件内引用的文件。一般来说，它们是包含图标、光标或者位图图象的二进制文件。

RC.EXE 资源编译器使用一个预处理器，这个预处理器可以进行常量的加减，把/*和*/作为注释表示，识别 C 的预处理指令#define 、#undef 、#ifdef 、#include 、#if 、#else 和#endif 。#include 指令与一般 C 程序稍有不同（我们将在第 11 章中详细讨论）。

使用资源时， make 文件中的另一个变化是.RES 文件（以及任何一个.OBJ 文件）也成了.EXE 目标的一个依赖文件。

9.2 图标和光标

让我们先来看一看一个使用图标和光标两种资源的示例程序。 RESOURC1 如图 9-1 所示，当程序窗口最小化时，它显示一个用户化的图标；当鼠标位于 RESOURC1 的客户区时，它使用一个定制的光标。 RESOURC1 还在客户区内画出几行和几列它的图标。

RESOURC1.MAK

```
#-----------------------
# RESOURC1.MAK make file
#-----------------------

resourc1.exe : resourc1.obj resourc1.res
     $(LINKER) $(GUIFLAGS) -OUT:resourc1.exe \
     resourc1.obj resourc1.res $(GUILIBS)

resourc1.obj : resourc1.c
     $(CC) $(CFLAGS) resourc1.c

resourc1.res : resourc1.rc resourc1.ico resourc1.cur
     $(RC) $(RCVARS) resourc1.rc
```

RESOURC1.C

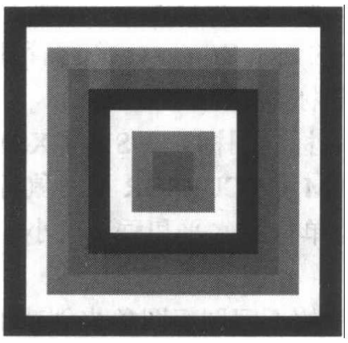
```
/*-----------------------------------------------------
   RESOURC1.C -- Icon and Cursor Demonstration Program No. 1
                 (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char      szAppName[] = "Resourc1" ;
HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (hInstance, szAppName) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, "Icon and Cursor Demo",
```

```
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HICON hIcon ;
     static int   cxIcon, cyIcon, cxClient, cyClient ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     int          x, y ;

     switch (iMsg)
          {
          case WM_CREATE :
               hIcon = LoadIcon (hInst, szAppName) ;
               cxIcon = GetSystemMetrics (SM_CXICON) ;
               cyIcon = GetSystemMetrics (SM_CYICON) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (y = cyIcon ; y < cyClient ; y += 2 * cyIcon)
                    for (x = cxIcon ; x < cxClient ; x += 2 * cxIcon)
                         DrawIcon (hdc, x, y, hIcon) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

RESOURC1.RC

```
/*-----------------------------
   RESOURC1.RC resource script
  -----------------------------*/

resourc1  ICON     resourc1.ico
resourc1  CURSOR   resourc1.cur
```

RESOURC1.ICO



RESOURC1.CUR

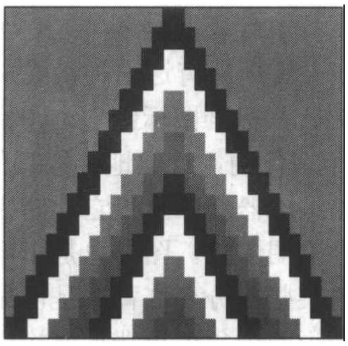


图 9-1　RESOURC1程序，包含一个图标和一个光标

你可以使用 Microsoft Developer Studio 或者 Windows 其它任何一种集成开发环境来创建图标和光标，它们以二进制格式保存。图标的扩展名为.ICO ，光标的扩展名为.CUR 。这些文件由 RESOURC1.RC 资源描述文件所引用。

9. 2. 1　图象编辑器

你用来创建图标、光标和位图的工具常被称为一个“图象编辑器”，这是任何一个 Windows 集成开发环境中最重要的开发工具之一。图标和光标都是位图的变体，所以先剖析一下位图将是很有用的。

位图是位的数组，一位或者多位对应于一个象素。在单色位图中，一位对应于一个象素。（在最简单的情况下，1 代表白色，0 代表黑色。然而，位图不只用于产生简单图象，还常常用于逻辑操作。）在彩色位图中，多个位对应于一个象素以表示颜色。图象编辑器支持生成单色位图和 16 色位图。在 16 色位图中，每个象素对应四位。

位图可以有多行多列，但是一些图象编辑器可能限制它们可以管理的位图大小。位图

储存在扩展名为.BMP 的文件中。你也可以在图象编辑器中创建图标和光标。

理论上，Windows 根据显示器的分辨率并且按照象素大小在屏幕上显示图标和光标，这样保证了图标和光标不会太大，也不会太小。但是，由于 VGA 视频适配器普遍采用 640 象素宽 480 象素高的屏幕大小，因此现在的图标和光标大小不象以前那么灵活可变了。

在 Windows 95 中，有两种大小的图标——标准的和小的。在一个 VGA 显示器上，标准图标为 32 象素点宽 32 象素点高，小图标是 16 象素点的正方形。小图标可以位于应用程序标题条的左边，用于激活系统菜单，也可以位于屏幕底部的系统任务条中，或者位于 Start 菜单中的程序列表里。出现在桌面上的图标是标准大小的。在 Windows 资源管理器中，以及在 Start 菜单中，用户可以有选择地挑选标准或者小图标。

程序可以用参数 SM_CXICON 和 SM_CYICON（对标准图标）、SM_CXSMICON 和 SM_CYSMICON（对小图标）、SM_CXCURSOR 和 SM_CYCURSOR（对鼠标光标）调用 GetSystemMetrics 函数来获取图标和光标的以象素为单位的水平和垂直大小。在大多数视频显示器上，标准图标和光标的大小是相同的。

Developer Studio 中包含的图象编辑器可以使用三种的不同图标图象来创建.ICO 文件：

- 标准：32 个象素的正方形，16 种颜色
- 单色：32 个象素的正方形，黑白
- 小：16 象素正方形，16 种颜色

在本章程序中显示的图标和光标都是单色的。你不必使用所有这三种格式来创建图标图象。在内部，Windows 可以通过简单地隔行隔列删除，从标准图标创建小图标。在单色显示器上（现在几乎完全过时了），Windows 可以使用灰色阴影近似地模拟颜色。

当你以三种格式中的一种来创建图标图象时，图象编辑器实际上将它存储为两个位图——一个是单色位图“掩码”，另一个是单色或彩色的位图图象。图标总是矩形的，但是掩码可以使图标看起来不是矩形的，也就是说，图标的某些部分允许背景是可见的。图标也可以包含反转背景颜色的区域。对于单色图标，下表说明图象编辑器如何构造用以描述图标的两种位图：

颜色	**黑色**	**白色**	**屏幕**	**反转屏幕**
掩码位图	0	0	1	1
图象位图	0	1	0	1

当显示图标时，Windows 首先将显示内容与第一个位图的各位进行按位“与”操作。第一个位图的 0 位对应的显示象素全为 0，代表黑色；对应于 1 位的显示象素保持不变，参见下表：

	显示象素	
掩码位	0	1
0	0	0
1	0	1

接着， Windows 对图象位图和显示内容进行按位“异或”操作。第二个位图中的 0 位保持显示象素不变，而 1 位反转显示象素颜色，下面是它的逻辑关系表：

	显示象素	
图象位	0	1
0	0	1
1	1	0

使用 C 语言的记法表示这些操作时，显示内容的变化遵循下列公式：

```
Display = (Display & Mask) ^Image
```

对于 16 色图标，掩码位图仍然是单色位图，其构造方法如上所述。图象位图中的每个象素包含 4 位以表示 16 种颜色，所有 4 位都置为 1 时表示反转背景的图标区域。

前面在讲述位图时曾经提到过， 0 并非一定是黑色， 1 也不意味着白色。正如你在这里所看到的一样，它依赖于 Windows 如何使用位图。

在 RESOURC1 中，我通过定义窗口类而将客户区的背景设置为 COLOR_WINDOW 。你可以在 Windows 控制面板中改变窗口颜色，来看一看图标和光标是如何反转颜色的。

9.2.2　获取图标句柄

资源描述文件使用如下所示的语句来引用图标文件：

```
myicon ICON iconfile.ico
```

其中， ICONFILE.ICO 是图标文件。这个语句把名字“ myicon ”分配给图标。在你的 C 程序中，使用 LoadIcon 函数来读取图标的句柄。 LoadIcon 需要两个参数，第一个参数是程序的实例句柄，即 WinMain 中的 hInstance 。 Windows 需要这个句柄来决定哪个.EXE 文件包含图标资源。第二个参数是指向 0 终结字符串的指针，该字符串是资源描述文件中的图标名。 LoadIcon 返回一个在 Windows.H 中定义的 HICON 类型的值。

下图说明资源描述文件中的图标名与 LoadIcon 语句之间的关系：

资源文件：	***myicon** ICON iconfile.ico*
程序源代码：	*hIcon = LoadIcon (hInstance, "**myicon**") ;*

这里不必担心字母的大小写，资源编译程序把资源描述文件名转换为大写，并把该名字插入到程序的.EXE 文件头部的资源表中。你第一次调用 LoadIcon 时， Windows 把第二个参数串转换为大写，并且在.EXE 文件的资源表中寻找相匹配的名字。

你还可以使用一个数字（一个 16 位的无符号 WORD ）来替代一个名字，这个数字被称为图标的 ID 号，方法如下：

资源描述文件： *125 ICON iconfile.ico*
程序源代码： *hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (125));*

MAKEINTRESOURCE （“将整数转换为资源串”）是 Windows 头文件中定义的一个宏，它将数字转换为一个指针，但是头 16 位被置为 0 。 Windows 由此知道 LoadIcon 的第二个参数是一个数字而非指向字符串的指针。

本书前面部分的示例程序使用预定义的图标：

```
LoadIcon (NULL, IDI_APPLICATION) ;
```

Windows 知道这是一个预定义图标，因为 hInstance 参数是 NULL 。 IDI_APPLICATION 也是在 WINDOWS 头文件中以 MAKEINTRESOURCE 定义的：

```
#define IDI_APPLICATION MAKEINTRESOURCE (32512) ;
```

预定义图标和光标是显示驱动程序文件的一部分。

你也可以使用字符串和数字相结合的另一种方法来引用图标名：

资源描述文件： *125 ICON iconfile.ico*
程序源代码： *hIcon = LoadIcon (hInstance, "#125");*

Windows 识别 # 字符开头的 ASCII 形式的数字。

那么第四种方法是什么呢？使用包含（直接使用#include ）在资源描述文件和程序的头文件中的宏定义：

头文件： *#define myicon 125*
资源描述文件： *myicon ICON iconfile.ico*
程序源代码： *hIcon = LoadIcon (hInstance, MAKEINTRESOURCE (myicon));*

当你使用这种方法时要小心，尽管图标名是字符串时其大小写无关紧要，但是由#define 语句产生的标识符是区分大小写的。

使用图标的 ID 号而不是图标名可以减小.EXE 文件的大小，加速 LoadIcon 调用。如果在程序中使用多个图标，你会发现将 ID 号放在数组中将使图标的使用变得更加方便。

9.2.3 在程序中使用图标

虽然 Windows 程序使用图标的方式有几种，但是许多 Windows 程序只是在定义窗口类时才指定图标：

```
wndclass.hIcon = LoadIcon (hInstance, "MyIcon") ;
.
```

```
wndclass.hIconSm = LoadIcon (hInstance, "MyIcon") ;
```

你可以在两个语句中引用相同标准大小的图标，而 Windows 会在显示小图标时调节大小。如果你以后想改变程序的图标，可以使用 SetClassLong 来做到这一点。让我们假设在你的资源描述文件中有另一个图标：

```
anothericon ICON iconfil2.ico
```

你可以通过下面的语句使这个图标取代 myicon 图标：

```
SetClassLong (hwnd, GCL_HICON,  LoadIcon (hInstance, "anothericon") ) ;
```

你可以使用 GCL_HICONSM 来改变小图标。如果你保存了 LoadIcon 调用所返回的图标句柄，那么你也可以在你窗口的客户区内画出图标：

```
DrawIcon (hdc, x, y, hIcon) ;
```

Windows 本身使用 DrawIcon 函数在图标区内显示程序的图标。 Windows 从窗口类结构中获得图标的句柄，你也可以使用同样的方法来获取图标的句柄：

```
DrawIcon (hdc, x, y, GetClassLong (hwnd, GCL_HICON) ;
```

RESOURC1 示例程序使用了同一图标说明窗口类，并且在窗口的客户区显示它。在资源描述文件中，图标与程序同名：

```
resourc1 ICON resourc1.ico
```

因为字符串“ Resourc1 ”被存放在数组 szAppName 中，并且已经用在程序中声明窗口类名，因此 LoadIcon 调用很简单：

```
LoadIcon (hInstance, szAppName) ;
```

你或许已经注意到，在 RESOURC1 中为同一图标调用了三次 LoadIcon ，两次是在 WinMain 中定义窗口类时，再一次是 WndProc 中处理 WM_CREATE 消息时获取图标句柄。三次调用 LoadIcon 不会有什么问题：两次调用都返回同样的句柄。 Windows 实际上只从.EXE 文件中装入一次图标。

9.2.4　使用其它的光标

你用来定义资源描述文件里的光标，以及在程序中获取光标句柄的语句同上面的图标语句非常相似：

资源描述文件：　　*mycursor CURSOR cursfile.cur*

程序代码：　　*hCursor = LoadCursor (hInstance, "mycursor") ;*

获取图标的其它方法(使用 ID 号和 MAKEINTRESOURCE)对光标也同样适用。Windows 头文件中包含了一个定义 HCURSOR 的 typedef 语句，可以用于存储光标句柄。

在设置窗口类结构的 hCursor 成员时，你可以用从 LoadCursor 中获得的光标句柄：

```
wndclass.hCursor = LoadCursor (hInstance, "mycursor") ;
```

这使得当鼠标出现在窗口的客户区时被显示为定制的光标。

如果你使用了子窗口，那么你可能希望光标随子窗口的不同而不同。如果你的程序为这些子窗口定义窗口类的话，那么就可以在每一个窗口类中适当地设置 hCursor 域，使得每一类都有不同的光标。如果使用了预定义子窗口控制，那么可以采用下列方法来改变窗口类的 hCursor 域：

```
SetClassLong (hwndChild, GCL_HCURSOR,
              LoadCursor (hInstance, "childcursor")) ;
```

如果你将客户区划分为较小的逻辑区域而不使用子窗口，那么你可以使用 SetCursor 语句来改变鼠标光标：

```
SetCursor (hCursor) ;
```

在处理 WM_MOUSEMOVE 消息期间，你应该调用 SetCursor，否则，当光标移动时，Windows 将使用窗口类中定义的光标来重画光标。

RESOURC1 使用程序名作为光标名：

```
resourc1 CURSOR resourc1.cur
```

当 RESOURC1.C 定义了窗口类时，LoadCursor 将使用 szAppName 变量：

```
wndclass.hCursor = LoadCursor (hInstance, szAppName) ;
```

9.2.5 位图：象素组成的图象

我们已经讨论过图标和光标中位图的使用，Windows 还包含一种叫做 BITMAP 的资源类型。

位图的使用主要有两个目的，第一是在显示器上显示图象。例如，Windows 显示驱动文件包括很多很小的位图，用于滚动条中的箭头、下拉式菜单中的选中标志、系统菜单框、放大缩小框，复选框以及单选按钮。你可以将位图用于菜单（正如在第 10 章中讨论的）和应用工具条（正如在第 12 章中讨论的）。例如 Paint 程序使用位图显示图形菜单。

位图的第二个主要用途是创建画刷，画刷是 Windows 用来填充显示区域的象素图案。

9.2.6 使用位图和画刷

RESOURC2 程序，如图 9-2 所示，这是由 RESOURC1 修改而成的。RESOURC1 包含一个用于在客户区的背景上创建画刷的单色位图资源，它是在图象编辑器中创建的 8x8 位图，这是画刷的最小尺寸。

RESOURC2.MAK

```
#-----------------------
# RESOURC2.MAK make file
#-----------------------

resourc2.exe : resourc2.obj resourc2.res
     $(LINKER) $(GUIFLAGS) -OUT:resourc2.exe \
     resourc2.obj resourc2.res $(GUILIBS)

resourc2.obj : resourc2.c
     $(CC) $(CFLAGS) resourc2.c

resourc2.res : resourc2.rc resourc2.ico resourc2.cur resourc2.bmp
     $(RC) $(RCVARS) resourc2.rc
```

RESOURC2.C

```
/*----------------------------------------------------------
   RESOURC2.C -- Icon and Cursor Demonstration Program No. 2
                 (c) Charles Petzold, 1996
  ----------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char      szAppName[] = "Resourc2" ;
HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HBITMAP    hBitmap ;
     HBRUSH     hBrush ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     hBitmap = LoadBitmap (hInstance, szAppName) ;
     hBrush = CreatePatternBrush (hBitmap) ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (hInstance, szAppName) ;
     wndclass.hbrBackground = hBrush ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;
```

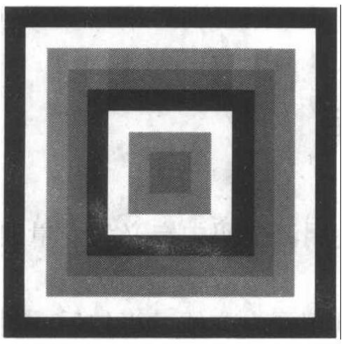
```
     RegisterClassEx (&wndclass) ;

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, "Icon and Cursor Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     DeleteObject ((HGDIOBJ) hBrush) ;       // clean-up
     DeleteObject ((HGDIOBJ) hBitmap) ;

     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HICON hIcon ;
     static int   cxIcon, cyIcon, cxClient, cyClient ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     int          x, y ;

     switch (iMsg)
          {
          case WM_CREATE :
               hIcon = LoadIcon (hInst, szAppName) ;
               cxIcon = GetSystemMetrics (SM_CXICON) ;
               cyIcon = GetSystemMetrics (SM_CYICON) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (y = cyIcon ; y < cyClient ; y += 2 * cyIcon)
                    for (x = cxIcon ; x < cxClient ; x += 2 * cxIcon)
                         DrawIcon (hdc, x, y, hIcon) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;
```

```
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

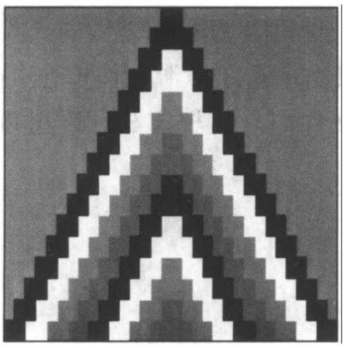
RESOURC2.RC

```
/*-------------------------------
   RESOURC2.RC resource script
  -------------------------------*/

resourc2  ICON    resourc2.ico
resourc2  CURSOR  resourc2.cur
resourc2  BITMAP  resourc2.bmp
```

RESOURC2.ICO



RESOURC2.CUR



RESOURC2.BMP

图 9-2 RESOURC2程序，包含一个图标、一个光标和一个位图

位图资源包含在与图标和光标格式相同的资源描述文件中：

```
resourc2 BITMAP resourc2.bmp
```

WinMain 中的 LoadBitmap 函数类似于 LoadIcon 和 LoadCursor 调用，它返回一个位图句柄：

```
hBitmap = LoadBitmap (hInstance, szAppName) ;
```

然后这个句柄可用于创建一个图案画刷，该画刷是基于该位图的：

```
hBrush = CreatePatternBrush (hBitmap) ;
```

当 Windows 使用该画刷填充显示区时，位图每隔八个象素就平行和垂直地重复一次。我们要用这个画刷给客户区的背景着色，这可以在定义窗口类时完成：

```
wndclass.hbrBackground = hBrush ;
```

位图和其它资源之间的主要区别实际上很简单，但是又很重要：位图是 GDI 对象。正确的程序结构要求，当不再需要它或在程序结束前，位图应该被删除。这一步在 RESOURC2 程序中的 WinMain 结束处进行：

```
DeleteObject (hBrush) ;
DeleteObject (hBitmap) ;
```

9.3 字符串

初步涉及字符串资源的用户似乎觉得有些奇怪，使用源代码中定义为变量的常规字符串时，我们并没有任何问题。

字符串资源主要是为了使程序转换成其它语言的过程更为方便。正如后面两章中看到的那样，菜单和对话框也是资源描述文件的一部分。如果使用字符串资源而不是将字符串直接放入源代码中，那么程序所使用的所有文本将在同一文件——资源描述文件中。如果转换了资源描述文件的文本，那么创建一个程序的另一种语言版本所需做的一切就是重新链接，以便把编译过的资源加到.EXE 文件中。这种方法比重新组织源代码安全得多（当然，你也可以将所有字符串定义为宏，并把它们存放在头文件中，这种方法同样可以避免在语言翻译过程中更新源代码）。

9.3.1 使用字符串资源

我们使用关键字 STRINGTABLE 在资源描述文件中定义字符串资源：

```
STRINGTABLE
    {
    id1, "Character string 1"
    id2, "Character string 2"
    [其它串定义]
    }
```

在资源描述文件中只包含一个字符串表，每个字符串只占一行，每一行最多 255 个字

符。字符串不能包含 C 格式控制字符，但\t（制表符）除外。然而，字符串可以包含八进制常量：

Tab	\011
换行	\012
回车	\013

这些控制字符由 DrawText 和 MessageBox 函数识别。

你的程序可以使用 LoadString 调用把字符串拷贝到程序数据段的缓冲区中去：

```
LoadString (hInstance, id, szBuffer, iMaxLength) ;
```

参数 id 是 ID 号，它加在资源描述文件中每个字符串的前面；szBuffer 是指向接收字符串的字符数组的指针；iMaxLength 是送入 szBuffer 中的最大字符数。每个字符串前面的 ID 值通常是定义在头文件中的宏标识符。许多 Windows 程序员喜欢使用前缀 IDS_来表示字符串的 ID 号。有时，文件名或者其它信息必须在显示字符串时嵌入到该字符串中，在这种情况下，你将 C 格式字符放入串中，并用 sprintf 或者 wsprintf 处理格式化字符串的方法来处理它们。

9.3.2 在消息框中使用字符串

让我们来看一个程序，它使用三个字符串，在消息框中显示三条错误信息。PROGRAM.H 头文件为这些信息定义了三个标识符：

```
#define IDS_FILENOTFOUND    1
#define IDS_FILETOOBIG      2
#define IDS_FILEREADONLY    3
```

资源描述文件格式如下：

```
#include "program.h"
[其它资源描述文件]
STRINGTABLE
    {
    IDS_FILENOTFOUND,   "File %s not found."
    IDS_FILETOOBIG,     "File %s too large to edit."
    IDS_FILEREADONLY,   "File %s is read-only."
    }
```

C 源代码文件也包含这个头文件，它定义了一个显示消息框的函数。（我假定 szAppName 是一个包含程序名的全局变量，而 hInst 是一个包含你程序的实例句柄的全局变量。）

```
#include "program.h"
[其它程序行]
```

```
OkMessage (HWND hwnd, int iErrorNumber, char *szFileName) ;
    {
    char szFormat[40];
    char szBuffer[60];

    LoadString (hInst, iErrorNumber, szFormat, 40) ;
    sprintf (szBuffer, szFormat, szFilename) ;

    return MessageBox (hwnd, szBuffer, szAppName,
                       MB_OK | MB_ICONEXCLAMATION) ;
    }
```

为了显示包含"file not found"信息的消息框，程序调用：

```
OkMessage (hwnd, IDS_FILENOTFOUND, szFileName) ;
```

9.4 用户定义的资源

“用户定义的资源”为连接.EXE文件中的各种数据提供了方便，对获取程序中的数据也是如此，数据可以按你的需要存放为文本格式或者二进制格式。当Windows将数据装入内存时，用于获取自定义资源的Windows函数将返回一个指向数据的指针。你可以对该数据进行任何处理，并会发现，与将数据存储在其它文件中然后使用文件输入函数进行访问比起来，这是存储和访问数据更方便的一种方法。

例如，假设你有一个叫做PROGHELP.TXT的文件，它包含程序的“help”文本。这个文件不一定是纯ASCII码文件，它也可以包含二进制数据，诸如可以帮助你的程序查找文件中不同段的指针。在你的资源文件中使用下面的语句调用该文件：

```
helptext TEXT proghelp.txt
```

这个语句中的名字helptext（资源名）和TEXT（资源类型）可以使用任意的名字。我将TEXT大写只是为了使它看起来和ICON、CURSOR以及BITMAP语句一样。我们在此所做的一切是为了生成自己的TEXT资源类型。

在程序的初始化（比如，在处理WM_CREATE消息时）期间，你可以获得资源的句柄：

```
hResource = LoadResource (hInstance,
            FindResource (hInstance, "TEXT", "helptext")) ;
```

变量hResource定义为HANDLE类型。不管它的名字是什么，LoadResource不会立即将资源装入内存。联合使用LoadResource和FindResource函数实质上类似于LoadIcon和LoadCursor函数。事实上，LoadIcon和LoadCursor函数用到了LoadResource和FindResource函数。

你可以用数字代替资源名和资源类型，这个数字可以由FindResource调用中的

MAKEINTRESOURCE 转换成远指针，用作资源类型的数字必须大于 255 。（当调用 FindResource 时 Windows 使用 1 到 9 之间的数字表示当前的资源类型。）

当你需要访问文本时，调用 LockResource:

```
pHelpText = LockResource (hResource) ;
```

LockResource 将资源装入内存（如果还没有装入的话），然后它返回一个指向资源的指针。当结束对资源的使用时，你可以从内存中释放它：

```
FreeResource (hResource) ;
```

当你的程序终止时，也会释放资源，即使你没有调用 FreeResource 。

让我们看一个使用三种资源——一个图标、一个字符串表和一个用户定义的资源——的示例程序。图 9-3 所示的 POEPOEM 程序在其客户区显示 Edgar Allan Poe 的 “ Annabel Lee ” 文本。用户定义的资源是文件 POEPOEM.ASC ，它包含了诗文，此文本文件以反斜杠（\）结束。

POEPOEM.MAK

```
#-----------------------
# POEPOEM.MAK make file
#-----------------------

poepoem.exe : poepoem.obj poepoem.res
     $(LINKER) $(GUIFLAGS) -OUT:poepoem.exe poepoem.obj poepoem.res $(GUILIBS)

poepoem.obj : poepoem.c poepoem.h
     $(CC) $(CFLAGS) poepoem.c

poepoem.res : poepoem.rc poepoem.ico poepoem.asc poepoem.h
     $(RC) $(RCVARS) poepoem.rc
```

POEPOEM.C

```
/*-------------------------------------------------
   POEPOEM.C -- Demonstrates User-Defined Resource
                (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>
#include "poepoem.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char      szAppName[10] ;
char      szCaption[35] ;
HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
```

```
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX wndclass ;

     LoadString (hInstance, IDS_APPNAME, szAppName, sizeof (szAppName)) ;
     LoadString (hInstance, IDS_CAPTION, szCaption, sizeof (szCaption)) ;

     wndclass.cbSize         = sizeof (wndclass) ;
     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc    = WndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName  = szAppName ;
     wndclass.hIconSm        = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, szCaption,
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))

          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char    *pText ;
     static HGLOBAL hResource ;
     static HWND    hScroll ;
     static int     iPosition, cxChar, cyChar, cyClient, iNumLInes, xScroll ;
     char           szPoemRes[15] ;
     HDC            hdc ;
     PAINTSTRUCT    ps ;
     RECT           rect ;
     TEXTMETRIC     tm ;

     switch (iMsg)
```

```
{
case WM_CREATE :
     hdc = GetDC (hwnd) ;
     GetTextMetrics (hdc, &tm) ;
     cxChar = tm.tmAveCharWidth ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;
     ReleaseDC (hwnd, hdc) ;

     xScroll = GetSystemMetrics (SM_CXVSCROLL) ;

     hScroll = CreateWindow ("scrollbar", NULL,
                    WS_CHILD | WS_VISIBLE | SBS_VERT,
                    0, 0, 0, 0,
                    hwnd, (HMENU) 1, hInst, NULL) ;

     LoadString (hInst, IDS_POEMRES, szPoemRes,
                         sizeof (szPoemRes)) ;

     hResource = LoadResource (hInst,
                  FindResource (hInst, szPoemRes, "TEXT")) ;

     pText = (char *) LockResource (hResource) ;

     iNumLInes = 0 ;

     while (*pText != '\\' && *pText != '\0')

          {
          if (*pText == '\n')
               iNumLInes ++ ;
          pText = AnsiNext (pText) ;
          }
     *pText = '\0' ;

     SetScrollRange (hScroll, SB_CTL, 0, iNumLInes, FALSE) ;
     SetScrollPos   (hScroll, SB_CTL, 0, FALSE) ;
     return 0 ;

case WM_SIZE :
     MoveWindow (hScroll, LOWORD (lParam) - xScroll, 0,
          xScroll, cyClient = HIWORD (lParam), TRUE) ;
     SetFocus (hwnd) ;
     return 0 ;

case WM_SETFOCUS :
     SetFocus (hScroll) ;
     return 0 ;

case WM_VSCROLL :
     switch (wParam)
          {
          case SB_TOP :
               iPosition = 0 ;
               break ;
          case SB_BOTTOM :
```

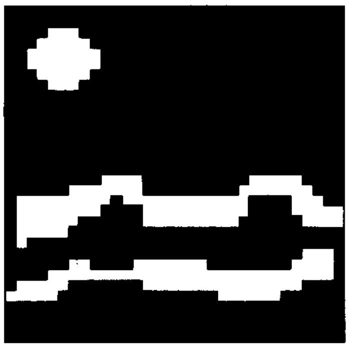
```
                    iPosition = iNumLInes ;
                    break ;
               case SB_LINEUP :
                    iPosition -= 1 ;
                    break ;
               case SB_LINEDOWN :
                    iPosition += 1 ;
                    break ;
               case SB_PAGEUP :
                    iPosition -= cyClient / cyChar ;
                    break ;
               case SB_PAGEDOWN :
                    iPosition += cyClient / cyChar ;
                    break ;
               case SB_THUMBPOSITION :
                    iPosition = LOWORD (lParam) ;
                    break ;

               }
          iPosition = max (0, min (iPosition, iNumLInes)) ;
          if (iPosition != GetScrollPos (hScroll, SB_CTL))
               {
               SetScrollPos (hScroll, SB_CTL, iPosition, TRUE) ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               }
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          pText = (char *) LockResource (hResource) ;

          GetClientRect (hwnd, &rect) ;
          rect.left += cxChar ;
          rect.top  += cyChar * (1 - iPosition) ;
          DrawText (hdc, pText, -1, &rect, DT_EXTERNALLEADING) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          FreeResource (hResource) ;
          PostQuitMessage (0) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

POEPOEM.RC

```
/*---------------------------
   POEPOEM.RC resource script
  ---------------------------*/

#include "poepoem.h"
```

```
poepoem       ICON   poepoem.ico
AnnabelLee    TEXT   poepoem.asc

STRINGTABLE
     {
     IDS_APPNAME,  "poepoem"
     IDS_CAPTION,  """Annabel Lee"" by Edgar Allen Poe"
     IDS_POEMRES,  "AnnabelLee"
     }
```

POEPOEM.ICO



POEPOEM.H

```
/*-----------------------
   POEPOEM.H header file
  -----------------------*/

#define IDS_APPNAME 0
#define IDS_CAPTION 1
#define IDS_POEMRES 2
```

POEPOEM.ASC

```
It was many and many a year ago,
   In a kingdom by the sea,
That a maiden there lived whom you may know
   By the name of Annabel Lee;
And this maiden she lived with no other thought
   Than to love and be loved by me.
I was a child and she was a child
   In this kingdom by the sea,
But we loved with a love that was more than love --
   I and my Annabel Lee --
With a love that the winged seraphs of Heaven
   Coveted her and me.
And this was the reason that, long ago,
   In this kingdom by the sea,
A wind blew out of a cloud, chilling
   My beautiful Annabel Lee;
So that her highborn kinsmen came
   And bore her away from me,
To shut her up in a sepulchre
```

```
     In this kingdom by the sea.
  The angels, not half so happy in Heaven,
     Went envying her and me --
  Yes! that was the reason (as all men know,
     In this kingdom by the sea)
  That the wind came out of the cloud by night,
     Chilling and killing my Annabel Lee.
  But our love it was stronger by far than the love
     Of those who were older than we --
     Of many far wiser than we --
  And neither the angels in Heaven above
     Nor the demons down under the sea
  Can ever dissever my soul from the soul
     Of the beautiful Annabel Lee:
  For the moon never beams, without bringing me dreams
    Of the beautiful Annabel Lee;
  And the stars never rise, but I feel the bright eyes
    Of the beautiful Annabel Lee:
  And so, all the night-tide, I lie down by the side
  Of my darling -- my darling -- my life and my bride,
     In her sepulchre there by the sea --
     In her tomb by the sounding sea.
                                            [May, 1849]
```

图 9-3 POEPOEM程序，包含一个图标和一个用户定义资源

在 POEPOEM.RC 资源描述文件中，用户定义的资源被定义为 TEXT 类型，取名为 AnnabelLee：

```
AnnabelLee TEXT poepoem.asc
```

在 WndProc 处理 WM_CREATE 时，使用 FindResource 和 LoadResource 获取资源句柄。使用 LockResource 锁定资源，并且使用一个小程序将文件末尾的反斜杠（\）换成 0，这有利于后面 WM_PAINT 消息期间使用的 DrawText 函数。

注意，这里使用的是子窗口的滚动条，而不是窗口滚动条，这是因为子窗口滚动条有一个自动的键盘接口，因此在 POEPOEM 中没有使用 WM_KEYDOWN 处理。

POEPOEM 还使用三个字符串，它们的 ID 号在 POEPOEM.H 头文件中定义。IDS_APPNAME 和 IDS_CAPTION 字符串由 LoadString 装入全局静态变量中：

```
LoadString (hInstance, IDS_APPNAME,szAppName, sizeof (szAppName)) ;
LoadString (hInstance, IDS_CAPTION, szCaption, sizeof (szCaption)) ;
```

既然我们已经定义了用于 POEPOEM 的所有字符串资源，那么就很容易让翻译者将程序转换成外语版本。当然，它们将不得不翻译文本"AnnabelLee"——我想，这会是一项困难得多的工作。

第 10 章　菜单和加速键

你还记得 Monty Python 有关奶酪店的幽默短剧吗？是这么一回事：一个客人走进奶酪店想买某种奶酪。当然，店里没有这种奶酪。因此他又问有没有另一种奶酪，然后再问另一种，再问另一种，再问另一种（最后总共问了 40 种的奶酪），但是回答仍然是：没有，没有，没有，没有，没有。

整个不幸事件可以通过菜单的使用来避免。一个菜单是一列可用的选项，它告诉饥饿的就餐者厨房可以提供哪些服务，并且——对于 Windows 程序来说——还告诉用户一个应用能够执行哪些操作。

菜单可能是 Windows 程序提供的一致用户界面的最重要部分，而在你的程序中增加菜单是 Windows 编程中相对简单的部分。你简单地在你的资源文件中定义菜单的结构并给每个菜单项赋予一个唯一的 ID 号。当用户选择一个菜单项时， Windows 给你的程序发送一个包含 ID 的 WM_COMMAND 消息。但是我们不会就此停留在这个简单的例子中，你可以对菜单做一件更有趣的事：在菜单中显示位图而非字符串，因此我们将仔细地看一下如何做到这一点。

本章还介绍了“键盘加速键”，这是一些键的组合，主要用于激活菜单功能。

10.1　菜　　单

窗口的菜单条紧接在标题条下方显示，这个菜单条有时被称为“主菜单”或者“顶层菜单”。列在顶层菜单的项通常是下拉式菜单，也叫做“弹出式菜单”或者“子菜单”。你也可以定义多重嵌套的弹出式菜单：也就是说，在弹出式菜单上的项可以访问另一个弹出式菜单。有时弹出式菜单中的项调用对话框以获得更多的信息（对话框将在下一章中介绍）。在标题条的最左端，很多父窗口都显示程序的小图标，这个图标激活系统菜单，这实际上是另一个弹出式菜单。

弹出式菜单的各项可以是“被选中的”，这意味着 Windows 在菜单文本的左端显示一个小的选中标记，选中标记让用户知道从菜单中选中了哪些选项。这些选项之间可以是互斥的，也可以不互斥。顶层菜单项不能被选中。

顶层菜单或者弹出式菜单项可以被“启用”、“禁用”或“灰化”。“激活”和“不激活”有时候被用作“启用”和“禁用”的同义词。标以启用或禁用的菜单项在用户看来是一样的，但是灰化的菜单项是使用灰色文本来显示的。

从用户的角度来看，启用、禁用和灰化的菜单项都是可以“选择的”（加亮的），也就是说，用户可以使用鼠标选择被禁用的菜单项，或者将反显光标条移动到禁用的菜单项

上，或者使用菜单项的关键字母来选择该菜单项。然而，从程序的角度来看，启用、禁用和灰化菜单项的功能是不同的。Windows 只为启用的菜单项向程序发送 WM_COMMAND 消息。对当前不合法的选项，可以禁用和灰化菜单项。如果你想使用户知道选择是不合法的，那么你可以灰化它。

10.1.1 菜单结构

当你创建或改变程序中的菜单时，把顶层菜单和每一个弹出式菜单想象成各自独立的菜单是有用的。顶层菜单有一个菜单句柄，在顶层菜单中的每一个弹出式菜单也有它自己的菜单句柄。系统菜单（也是一个弹出式菜单）也有一个菜单句柄。

菜单中的每一项都有三个特性。第一个特性是菜单中显示什么，它可以是字符串或者一个位图。第二个特性是在 WM_COMMAND 消息中 Windows 发送给程序的菜单 ID，或者在用户选择菜单项时 Windows 显示的弹出式菜单是什么。第三个特性是菜单项的属性，包括菜单项是否被禁用、灰化或者是被选中。

10.1.2 菜单模板

你可以使用三种不同的方式创建菜单。最通用（也是最简单）的方法是在资源描述文件中以菜单模板的形式定义菜单，例如：

```
MyMenu MENU
     {
     [菜单列表]
     }
```

MyMenu 是菜单的名字，你在窗口类结构中引用这个名字，菜单名通常和应用程序的名字一样。

在花括号中，你可以使用 MENUITEM 或者 POPUP 语句。MENUITEM 语句的格式如下：

```
MENUITEM "&Text", id [, options]
```

POPUP 语句的格式如下：

```
POPUP "&Text" [, options]
     {
     [菜单列表]
     }
```

也可以不使用花括号，而使用 BEGIN 和 END 。每一个菜单显示的文本必须在一对双引号中。当 Windows 显示菜单时，&号导致紧跟其后的字符带下划线，该字符也是在使用 Alt 键选择菜单项时 Windows 所要查找的字符。如果你在文本中没有符号&，那么就不会出现

下划线，并且在用 Alt 键查找时 Windows 将使用菜单文本的第一个字母。

在顶层菜单中， MENUITEM 和 POPUP 语句行中所加的各种选项如下所示：

- GRAYED ——菜单项不是活动的，并且不产生 WM_COMMAND 消息。文本被灰化。
- INACTIVE ——菜单项不是活动的，并且不产生 WM_COMMAND 消息。文本正常显示。
- MENUBREAK ——这一项和后随的项显示在新的菜单行中。
- HELP ——这一项和后随的项是右对齐的。

使用 C 的按位“或”（|）操作可以将各项组合在一起，但是不能一起使用 GRAYED 和 INACTIVE 。在顶层菜单中，不常使用 MENUBREAK ，因为如果窗口太窄而不能放下整个菜单时， Windows 会自动地把顶层菜单分成多行。

在主菜单中的 POPUP 语句之后，左右括号（或者 BEGIN 和 END 关键字）将弹出式菜单中的各项分隔开。在弹出式菜单的定义中，允许使用下列语句：

```
MENUITEM "text", id [, options]
```

和：

```
MENUITEM SEPARATOR
```

和：

```
POPUP "text" [, options]
```

MENUITEM SEPARATOR 在弹出式菜单中画一水平线，通常用于把相关的选项组分开。

对于弹出式菜单中的各项，你可以在字符串中使用制表符\t ，随后的文本放在一个新的列中，该列在左边留出足够的空间以便容纳弹出式菜单中第一列里最长的文本字符串。我们将在本章后面讨论加速键时，看到这是怎样工作的。\a 将紧跟在其后的文本右对齐，弹出式菜单的 MENUITEM 选项如下：

- CHECKED——文本左端出现选中标志
- CRAYED——菜单项不是活动的，并且不产生 WM_COMMAND 消息。文本灰化。
- INACTIVE——菜单项不是活动的，并且不产生 WM_COMMAND 消息。文本正常显示。
- MENUBREAK——这个项和随后的项出现在新的菜单列中。
- MENUBARBREAK——这个项和随后的项出现在新的菜单列中，竖线将各列分开。

不能联合使用 GRAYED 和 INACTIVE ，也不能联合使用 MENUBREAK 和 MENUBARBREAK 。在弹出式菜单中项数太多而不能显示在单独一列时，你应该使用 MENUBREAK 或者 MENUBARBREAK 。

MENUITEM 语句中的 id 值是 Windows 在菜单消息中发送给窗口过程的值。在一个菜单中， id 值应该是唯一的。如果不使用数值，你可以使用头文件中定义的标识符。习惯上，这些标识符以字母 IDM （“表示用于菜单的 ID ”）开头。

10.1.3 在程序中引用菜单

多数 Windows 应用程序的资源描述文件中只包含一个菜单，程序在窗口类的定义中引用这个菜单：

```
wndclass.lpszMenuName = "MyMenu";
```

程序员经常使用程序名作为菜单名，因此同样的字符串也可以用于窗口类、程序图标名和菜单名。然而，你也可以为菜单使用一个数值（或者宏定义标识符）而不使用名称。资源描述文件如下所示：

```
45 MENU
    {
    [菜单定义]
    }
```

在这种情况下，对窗口类结构的 lpszMenuName 域赋值的语句可以是：

```
wndclass.lpszMenuName = MAKEINTRESOURCE (45) ;
```

或者：

```
wndclass.lpszMenuName = "#45";
```

虽然访问菜单资源的最常用的方法是在窗口类中指定菜单，你也可以使用其它方法。Windows 应用可以使用 LoadMenu 函数将菜单资源装入内存中，如同在第 9 章中描述的 LoadIcon 和 LoadCursor 函数一样。你可以在资源描述文件中使用菜单名，那么 LoadMenu 将返回一个菜单句柄：

```
hMenu = LoadMenu (hInstance, "MyMenu") ;
```

如果你使用一个数值，那么 LoadMenu 将采用如下形式：

```
hMenu = LoadMenu (hInstance,  MAKEINTRESOURCE (45)) ;
```

或者：

```
hMenu = LoadMenu (hInstance, "#45") ;
```

然后，你可以将这个菜单句柄作为 CreateWindow 的第九个参数：

```
hwnd = CreateWindow ("MyClass", "Window Caption",
            WS_OVERLAPPEDWINDOW,
            WS_USEDEFAULT, CW_USEDEFAULT,
            CW_USEDEFAULT, CW_USEDEFAULT,
```

```
NULL,
hMenu,
hInstance,
NULL) ;
```

在这种情况下， CreateWindow 调用中指定的菜单可以覆盖窗口类中指定的任何菜单。如果 CreateWindow 的第九个参数是 NULL，那么你可以把窗口类中的菜单看作是基于这个窗口类的窗口缺省菜单。这样，你可以为基于同一窗口类的几个窗口使用不同的菜单。

你也可以在窗口类中指定 NULL 菜单，并且在 CreateWindow 调用中也指定 NULL 菜单，然后在窗口被创建后再给窗口指定一个菜单：

```
SetMenu (hwnd, hMenu) ;
```

这种形式使你可以动态地修改窗口的菜单。在本章后面的 NOPOPUPS 程序中我们将会看到这方面的例子。

当窗口被清除时，同窗口相关的所有菜单都将被清除。同窗口不相关的菜单在程序结束前通过调用 DestroyMenu 被显示地清除。

10.1.4　菜单和消息

当用户选择一个菜单项时， Windows 通常向窗口过程发送几个不同的消息。在大多数情况下，你的程序可以忽略大部分消息，只需把它们传递给 DefWindowProc 即可。WM_INITMENU 就是这一类的消息，它具有下列参数：

wParam	lParam
主菜单句柄	0

wParam 值是你的主菜单句柄，即使用户选择的是系统菜单中的项。 Windows 程序通常忽略 WM_INITMENU 消息。尽管在选中该项之前的消息已经给程序提供了修改菜单的机会，但是我们觉得此刻改变顶层菜单对用户来说是很为难的。

程序也接收 WM_MENUSELECT 消息。随着用户在菜单项中移动光标或者鼠标，程序会收到许多 WM_MENUSELECT 消息。而正如我们将在第 12 章中看到的，这对实现那些常有对菜单项的文本描述的状态条是很有帮助的。 WM_MENUSELECT 的参数如下所示：

LOWORD (wParam)	HIWORD (wParam)	lParam
选中项：菜单 ID 或者弹出式菜单句柄	选择标志	包含被选中项的菜单句柄

WM_MENUSELECT 是一个菜单跟踪消息， wParam 的低位字告诉你当前选择的是菜单中的哪一项（加亮）， wParam 的高位字中的“选择标志”可以是下列这些标志的组合：MF_GRAYED 、 MF_DISABLED 、 MF_CHECKED 、 MF_BITMAP 、 MF_POPUP, MF_HELP 、 MF_SYSMENU 和 MF_MOUSESELECT 。如果你需要根据对菜单项的选择来

改变窗口客户区的内容，那么你可以使用 WM_MENUSELECT 消息。许多程序把该消息发送给 DefWindowProc 。

当 Windows 准备显示一个弹出式菜单时，它给窗口过程发送一条 WM_INITMENUPOPUP 消息，参数如下：

wParam	LOWORD (lParam)	HIWORD (lParam)
弹出式菜单句柄	弹出式菜单索引	系统菜单为 1 ，其它为 0

如果你需要在显示弹出式菜单之前启用或者禁用菜单项，那么这条消息就很重要。例如，假定程序使用弹出式菜单上的 Paste 命令从剪贴板拷贝文本，当你收到弹出式菜单中的 WM_INITMENUPOPUP 消息时，应确定剪贴板内是否有文本存在。如果没有，那么应该使 Paste 菜单项灰化。我们将在本章后面修改的 POPPAD 程序中看到这样的例子。

最重要的菜单消息是 WM_COMMAND ，它表明用户已经从菜单中选中了一个被启用的菜单项。第 8 章中的 WM_COMMAND 消息也可以由子窗口控制产生。如果你碰巧为菜单和子窗口控制使用同一 ID 码，那么你可以通过 lParam 的低位字来区别它们，菜单项的 lParam 值为 0 ：

	LOWORD (wParam)	HIIWORD (wParam)	lParam
菜单：	菜单 ID	0	0
控制：	控制 ID	通知码	子窗口句柄

WM_SYSCOMMAND 消息类似于 WM_COMMAND 消息，只是 WM_SYSCOMMAND 表示用户从系统菜单中选择一个启用的菜单项：

	LOWORD (wParam)	HIWORD (wParam)	lParam
系统菜单：	菜单 ID	0	0（如果 WM_SYSCOMMAND 消息是按动鼠标的结果，那么 LOWORD (lParam) 和 HIWORD (lParam)分别表示鼠标光标的 X 和 Y 屏幕坐标。）

菜单 ID 指示系统菜单中的哪一项被选中。对于预定义的系统菜单项，应该屏蔽低四位，结果值应该为下列之一： SC_SIZE 、 SC_MOVE 、 SC_MINIMIZE 、 SC_MAXIMIZE 、 SC_NEXTWINDOW 、 SC_PREVWINDOW 、 SC_CLOSE 、 SC_VSCROLL 、 SC_HSCROLL 、 SC_ARRANGE 、 SC_RESTORE 和 SC_TASKLIST 。此外， wParam 的低位字可以是 SC_MOUSEMENU 或者 SC_KEYMENU 。

如果你在系统菜单中添加菜单项，那么 wParam 的低位字将是你定义的菜单 ID 。为了避免与预定义的菜单 ID 相冲突，应用程序应该使用小于 F000H 的值，这对于将一般的 WM_SYSCOMMAND 消息发送给 DefWindowProc 是很重要的。如果你不这样做，那么你实际上就是禁用了正常的系统菜单命令。

我们将讨论的最后一条消息是 WM_MENUCHAR，它实际上根本不是菜单消息。在下列两种情况之一发生时，Windows 会把这条消息发送到窗口过程：如果用户按 Alt 和一个与菜单项不匹配的字符时，或者当显示弹出式菜单而用户按下一个与弹出式菜单里的项不匹配的字符键时。随 WM_MENUCHAR 消息一起发送的参数如下所示：

LOWORD (wParam)	**HIWORD (wParam)**	**lParam**
ASCII 码	选择码	菜单句柄

选择码是：

- 0——不显示弹出式菜单
- MF_POP——显示弹出式菜单
- MF_SYSMENU——显示系统弹出式菜单

Windows 程序通常把这条消息传送给 DefWindowProc，DefWindowProc 一般返回一个 0 给 Windows，这使得 Windows 发出蜂鸣声。我们将在本章后面的 GRAFMENU 程序中看到 WM_MENUCHAR 消息的应用。

10.1.5 示例程序

让我们来看一个简单的例子。图 10-1 所示的 MENUDEMO 程序在主菜单中有 5 项——File、Edit、Backgroud、Timer 和 Help，每一项有一个弹出式菜单。MENUDEMO 只完成了最简单、最通用的菜单处理操作，包括捕获 WM_COMMAND 消息和检查 wParam 的低位字。

MENUDEMO.MAK

```
#-----------------------
# MENUDEMO.MAK make file
#-----------------------

menudemo.exe : menudemo.obj menudemo.res
     $(LINKER) $(GUIFLAGS) -OUT:menudemo.exe menudemo.obj \
     menudemo.res $(GUILIBS)

menudemo.obj : menudemo.c menudemo.h
     $(CC) $(CFLAGS) menudemo.c

menudemo.res : menudemo.rc menudemo.h
     $(RC) $(RCVARS) menudemo.rc
```

MENUDEMO.C

```
/*----------------------------------------
   MENUDEMO.C -- Menu Demonstration
                 (c) Charles Petzold, 1996
  ----------------------------------------*/
```

```
#include <windows.h>
#include "menudemo.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[] = "MenuDemo" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Menu Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int  iColorID[5] = { WHITE_BRUSH,  LTGRAY_BRUSH, GRAY_BRUSH,
                                 DKGRAY_BRUSH, BLACK_BRUSH } ;
     static int  iSelection = IDM_WHITE ;
     HMENU       hMenu ;

     switch (iMsg)
          {
```

```
case WM_COMMAND :
     hMenu = GetMenu (hwnd) ;

     switch (LOWORD (wParam))
          {
          case IDM_NEW :
          case IDM_OPEN :
          case IDM_SAVE :
          case IDM_SAVEAS :
               MessageBeep (0) ;
               return 0 ;

          case IDM_EXIT :
               SendMessage (hwnd, WM_CLOSE, 0, 0L) ;
               return 0 ;

          case IDM_UNDO :
          case IDM_CUT :
          case IDM_COPY :
          case IDM_PASTE :
          case IDM_DEL :
               MessageBeep (0) ;
               return 0 ;

          case IDM_WHITE :          // Note: Logic below
          case IDM_LTGRAY :         //   assumes that IDM_WHITE
          case IDM_GRAY :           //   through IDM_BLACK are
          case IDM_DKGRAY :         //   consecutive numbers in
          case IDM_BLACK :          //   the order shown here.

               CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
               iSelection = LOWORD (wParam) ;
               CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;

               SetClassLong (hwnd, GCL_HBRBACKGROUND,
                    (LONG) GetStockObject
                       (iColorID[LOWORD (wParam) - IDM_WHITE])) ;

               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case IDM_START :
               if (SetTimer (hwnd, 1, 1000, NULL))
                    {
                    EnableMenuItem (hMenu, IDM_START, MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_STOP, MF_ENABLED) ;
                    }
               return 0 ;

          case IDM_STOP :
               KillTimer (hwnd, 1) ;
               EnableMenuItem (hMenu, IDM_START, MF_ENABLED) ;
               EnableMenuItem (hMenu, IDM_STOP,  MF_GRAYED) ;
               return 0 ;
```

```
                    case IDM_HELP :
                         MessageBox (hwnd, "Help not yet implemented!",
                                     szAppName, MB_ICONEXCLAMATION | MB_OK) ;
                         return 0 ;

                    case IDM_ABOUT :
                         MessageBox (hwnd, "Menu Demonstration Program.",
                                     szAppName, MB_ICONINFORMATION | MB_OK) ;
                         return 0 ;
                    }
               break ;

          case WM_TIMER :
               MessageBeep (0) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

MENUDEMO.RC

```
/*-----------------------------
   MENUDEMO.RC resource script
  -----------------------------*/

#include "menudemo.h"

MenuDemo MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New",                   IDM_NEW
          MENUITEM "&Open...",               IDM_OPEN
          MENUITEM "&Save",                  IDM_SAVE
          MENUITEM "Save &As...",            IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo",                  IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t",                   IDM_CUT
          MENUITEM "&Copy",                  IDM_COPY
          MENUITEM "&Paste",                 IDM_PASTE
          MENUITEM "De&lete",                IDM_DEL
          }
     POPUP "&Background"
          {
          MENUITEM "&White",                 IDM_WHITE, CHECKED
          MENUITEM "&Lt Gray",               IDM_LTGRAY
```

```
          MENUITEM "&Gray",              IDM_GRAY
          MENUITEM "&Dk Gray",           IDM_DKGRAY
          MENUITEM "&Black",             IDM_BLACK
          }
     POPUP "&Timer"
          {
          MENUITEM "&Start"              IDM_START
          MENUITEM "S&top"               IDM_STOP,  GRAYED
          }
     POPUP "&Help"
          {
          MENUITEM "&Help...",           IDM_HELP
          MENUITEM "&About MenuDemo...", IDM_ABOUT
          }
     }
```

MENUDEMO.H

```
/*------------------------
   MENUDEMO.H header file
  ------------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4
#define IDM_EXIT     5

#define IDM_UNDO    10
#define IDM_CUT     11
#define IDM_COPY    12
#define IDM_PASTE   13
#define IDM_DEL     14

#define IDM_WHITE   20
#define IDM_LTGRAY  21
#define IDM_GRAY    22
#define IDM_DKGRAY  23
#define IDM_BLACK   24

#define IDM_START   30
#define IDM_STOP    31

#define IDM_HELP    40
#define IDM_ABOUT   41
```

图 10-1 MENUDEMO程序

所有菜单 ID 的标识符都定义在 MENUDEMO.H 中，这个文件必须（用#include 语句）在资源描述文件和 C 源代码文件中指定。标识符以 IDM 打头。为菜单定义的 ID 值不一定是连续的，然而，如果你使用 switch 和 case 语句来处理程序中的 ID 值，那么请记住，采用连续的菜单 ID 值时， C 编译程序就可以使用跳转表来最大限度地优化代码。

当接收到与弹出式菜单 File 和 Edit 各项有关的 WM_COMMAND 时， MENUDEMO

程序只使系统发出蜂鸣声。Background 弹出式菜单列出 MENUDEMO 用来给背景着色的五种现有画刷。在 MENUDEMO.RC 资源描述文件中，White 菜单项（菜单 ID 为 IDM_WHITE）被标以 CHECKED，它在菜单项旁边设置一选中标记。在 MENUDEMO.C 中，iSelection 的值被初始化为 IDM_WHITE。

Background 弹出式菜单上的 5 种画刷相互排斥。当 MENUDEMO.C 收到一条 WM_COMMAND 消息，而该消息中 wParam 的低位字是 Background 弹出式菜单上的 5 项之一时，它必须从先前选中的背景颜色中除掉选中标记，并把标记加到新的背景颜色上。为此，首先要得到菜单句柄：

```
hMenu = GetMenu (hwnd) ;
```

CheckMenuItem 函数用来取消当前被选中的项。

```
CheckMenuItem (hMenu, iSelection, MF_UNCKECKED) ;
```

iSelection 的值被设置为 wParam 的低位字，新的背景颜色被选中：

```
iSelection = LOWORD (wParam) ;
CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;
```

窗口类中的背景颜色于是被替换为新的背景颜色，窗口客户区变为无效状态，Windows 使用新的背景颜色清除窗口。

Timer 弹出式菜单列出了两个选项——Start 和 Stop。开始时，Stop 选项被变灰（就像在资源描述文件中的菜单定义一样）。当你选择 Start 选项时，MENUDEMO 试图启动一个计时器，如果成功，则灰化 Start 选项，并激活 Stop 选项：

```
EnableMenuItem (hMenu, IDM_START, MF_GRAYED) ;
EnableMenuItem (hMenu, IDM_STOP, MF_ENABLED) ;
```

当收到一条 WM_COMMAND 消息，并且 wParam 的低位字等于 IDM_STOP 时，MENUDEMO 程序停止计数，激活 Start 项，灰化 Stop 选项：

```
EnableMenuItem (hMenu, IDM_START, MF_ENABLED) ;
EnableMenuItem (hMenu, IDM_STOP, MF_GRAYED) ;
```

请注意，在计时器运行时，MENUDEMO 程序不可能收到 wParam 低位字等于 IDM_START 的 WM_COMMAND 消息。与此类似，在计时器关闭时，MENUDEMO 程序也不可能收到 wParam 低位字等于 IDM_STOP 的 WM_COMMAND 消息。

当 MENUDEMO 收到一条 WM_COMMAND 消息，而该消息的参数 wParam 的低位字等于 IDM_ABOUT 或 IDM_HELP 时，MENUDEMO 程序显示一个消息框（在下一章中，我们将把消息框变为对话框）。

当 MENUDEMO 程序收到一条 WM_COMMAND 消息，其参数 wParam 的低位字等于 IDM_EXIT 时，它给自己发送一条 WM_CLOSE 消息。这条消息与 DefWindowProc 收到 WM_SYSCOMMAND 消息且 wParam 等于 SC_CLOSE 时发送给窗口过程的消息相同。我们将在本章后面介绍 POPPAD2 时再仔细研究这个问题。

10.1.6 菜单设计规范

在 MENUDEMO 中的 File 和 Edit 弹出式菜单的格式与其它程序中的格式非常类似。Windows 的目的之一是为用户提供一种可识别的界面，而不要求用户为每个程序重新学习基本概念。如果 File 和 Edit 菜单在每个 Windows 程序中看起来都一样，并且都使用同样的字母和 Alt 键来进行选择，那么当然不会需要什么帮助了。

除了 File 和 Edit 弹出式菜单外，大多数 Windows 程序的菜单都是不同的。当设计一个菜单时，你应该看一看现有的 Windows 程序以尽量保持一致。当然，如果你认为别的程序是不对的，而你知道正确的方法，那么没有人能够阻止你。同时记住，修改一个菜单，通常只需要修改资源描述文件而不必修改你的程序代码。即使以后要改变菜单项的位置，也不会有多大的问题。

虽然你的程序菜单在顶层可以有 MENUITEM 语句，但这是不规范的，因为这样会很容易导致错误的选择。如果你这样做，那么在文本串后面用一个惊叹号表示菜单项不会激活弹出式菜单。

10.1.7 定义菜单的复杂方法

在程序的资源描述文件中定义菜单通常是在你的窗口中添加菜单的最简单方法，但不是唯一的方法。如果你没有使用资源描述文件，那么可以使用 CreateMenu 和 ChangeMenu 两个函数在程序中创建菜单。在你定义完菜单后，你可以将菜单句柄发送给 CreateWindow 或者使用 SetMenu 来设置窗口的菜单。

以下是具体的做法。CreateMenu 返回一个新的菜单句柄：

```
hMenu = CreateMenu () ;
```

菜单一开始为空，AppendMenu 将菜单项插入菜单中。你必须为顶层菜单项和每一个弹出式菜单提供不同的菜单句柄。弹出式菜单是单独构成的；然后将弹出式菜单句柄插入顶层菜单。图 10-2 中所示的代码就是用这种方法创建菜单的；实际上，这个菜单与 MENUDEMO 程序中的菜单相同。

```
hMenu = CreateMenu () ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,     IDM_NEW,     "&New") ;
AppendMenu (hMenuPopup, MF_STRING,     IDM_OPEN,    "&Open...") ;
AppendMenu (hMenuPopup, MF_STRING,     IDM_SAVE,    "&Save") ;
AppendMenu (hMenuPopup, MF_STRING,     IDM_SAVEAS,  "Save &As...") ;
AppendMenu (hMenuPopup, MF_SEPARATOR, 0,           NULL) ;
AppendMenu (hMenuPopup, MF_STRING,     IDM_EXIT,    "E&xit") ;

AppendMenu (hMenu, MF_POPUP, (UINT) hMenuPopup, "&File") ;
```

```
hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,    IDM_UNDO,  "&Undo") ;
AppendMenu (hMenuPopup, MF_SEPARATOR, 0,         NULL) ;
AppendMenu (hMenuPopup, MF_STRING,    IDM_CUT,   "Cu&t") ;
AppendMenu (hMenuPopup, MF_STRING,    IDM_COPY,  "&Copy") ;
AppendMenu (hMenuPopup, MF_STRING,    IDM_PASTE, "&Paste") ;
AppendMenu (hMenuPopup, MF_STRING,    IDM_DEL,   "De&lete") ;

AppendMenu (hMenu, MF_POPUP, (UINT) hMenuPopup, "&Edit") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING | MF_CHECKED, IDM_WHITE,  "&White") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_LTGRAY, "&Lt Gray") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_GRAY,   "&Gray") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_DKGRAY, "&Dk Gray") ;
AppendMenu (hMenuPopup, MF_STRING,              IDM_BLACK,  "&Black") ;

AppendMenu (hMenu, MF_POPUP, (UINT) hMenuPopup, "&Background") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING,             IDM_START, "&Start") ;
AppendMenu (hMenuPopup, MF_STRING | MF_GRAYED, IDM_STOP,  "S&top") ;
AppendMenu (hMenu, MF_POPUP, (UINT) hMenuPopup, "&Timer") ;

hMenuPopup = CreateMenu () ;

AppendMenu (hMenuPopup, MF_STRING, IDM_HELP,  "&Help...") ;
AppendMenu (hMenuPopup, MF_STRING, IDM_ABOUT, "&About MenuDemo...") ;

AppendMenu (hMenu, MF_POPUP, (UINT) hMenuPopup, "&Help") ;
```

图 10-2 不使用资源描述文件创建与MENUDEMO程序相同菜单的C代码

我认为你会同意这个观点，即，使用资源描述文件菜单模板更容易更清楚一些。我并不鼓励你使用这里的方法定义菜单，而只是提供了一种实现菜单的方法。当然，你可以使用包含所有菜单项字符串、ID 和标志等的结构数组来压缩代码大小。不过，如果你这么做了，那么你还可以利用 Windows 定义菜单的第三种方法。

10.1.8 定义菜单的第三种方法

LoadMenuIndirect 函数接受一个指向 MENUITEMTEMPLATE 类型的结构指针，并返回菜单的句柄，该函数在装入资源描述文件中的常规菜单模板后在 Windows 中构造菜单，读者不妨自己尝试一下。

10.1.9 浮动弹出式菜单

你还可以在没有顶层菜单条的情况下使用菜单，也就是说，你可以使弹出式菜单出现在屏幕顶层的任何位置。一种方法是使用右鼠标按钮来激活弹出式菜单，然而，菜单项往往还是由左鼠标按钮来进行选择的。图 10-3 所示的 POPMENU 说明了这种方法。

POPMENU.MAK

```
#-----------------------
# POPMENU.MAK make file
#-----------------------

popmenu.exe : popmenu.obj popmenu.res
     $(LINKER) $(GUIFLAGS) -OUT:popmenu.exe popmenu.obj \
     popmenu.res $(GUILIBS)

popmenu.obj : popmenu.c popmenu.h
     $(CC) $(CFLAGS) popmenu.c

popmenu.res : popmenu.rc popmenu.h
     $(RC) $(RCVARS) popmenu.rc
```

POPMENU.C

```
/*----------------------------------------
   POPMENU.C -- Popup Menu Demonstration
                (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include "popmenu.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char      szAppName[] = "PopMenu" ;
HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
```

```
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, "Popup Menu Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HMENU hMenu ;
     static int   iColorID[5] = { WHITE_BRUSH,  LTGRAY_BRUSH, GRAY_BRUSH,
                                  DKGRAY_BRUSH, BLACK_BRUSH } ;
     static int   iSelection = IDM_WHITE ;
     POINT        point ;

     switch (iMsg)
          {
          case WM_CREATE :
               hMenu = LoadMenu (hInst, szAppName) ;
               hMenu = GetSubMenu (hMenu, 0) ;
               return 0 ;

          case WM_RBUTTONDOWN :
               point.x = LOWORD (lParam) ;
               point.y = HIWORD (lParam) ;
               ClientToScreen (hwnd, &point) ;

               TrackPopupMenu (hMenu, 0, point.x, point.y, 0, hwnd, NULL) ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
```

```
                case IDM_SAVEAS :
                case IDM_UNDO :
                case IDM_CUT :
                case IDM_COPY :
                case IDM_PASTE :
                case IDM_DEL :
                     MessageBeep (0) ;
                     return 0 ;

                case IDM_WHITE :          // Note: Logic below
                case IDM_LTGRAY :         //   assumes that IDM_WHITE
                case IDM_GRAY :           //   through IDM_BLACK are
                case IDM_DKGRAY :         //   consecutive numbers in
                case IDM_BLACK :          //   the order shown here.

                     CheckMenuItem (hMenu, iSelection, MF_UNCHECKED) ;
                     iSelection = LOWORD (wParam) ;
                     CheckMenuItem (hMenu, iSelection, MF_CHECKED) ;

                     SetClassLong (hwnd, GCL_HBRBACKGROUND,
                          (LONG) GetStockObject
                             (iColorID[LOWORD (wParam) - IDM_WHITE])) ;

                     InvalidateRect (hwnd, NULL, TRUE) ;
                     return 0 ;

                case IDM_ABOUT :
                     MessageBox (hwnd,
                                 "Popup Menu Demonstration Program.",
                                 szAppName,
                                 MB_ICONINFORMATION | MB_OK) ;
                     return 0 ;

                case IDM_EXIT :
                     SendMessage (hwnd, WM_CLOSE, 0, 0) ;
                     return 0 ;

                case IDM_HELP :
                     MessageBox (hwnd, "Help not yet implemented!",
                                 szAppName,
                                 MB_ICONEXCLAMATION | MB_OK) ;
                     return 0 ;
                }
           break ;

      case WM_DESTROY :
           PostQuitMessage (0) ;
           return 0 ;
      }
 return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
 }
```

POPMENU.RC

```
/*---------------------------
   POPMENU.RC resource script
  ---------------------------*/

#include "popmenu.h"

PopMenu MENU
     {
     POPUP ""
          {
          POPUP "&File"
               {
               MENUITEM "&New",              IDM_NEW
               MENUITEM "&Open...",          IDM_OPEN
               MENUITEM "&Save",             IDM_SAVE
               MENUITEM "Save &As...",       IDM_SAVEAS
               MENUITEM SEPARATOR
               MENUITEM "E&xit",             IDM_EXIT
               }
          POPUP "&Edit"
               {
               MENUITEM "&Undo",             IDM_UNDO
               MENUITEM SEPARATOR
               MENUITEM "Cu&t",              IDM_CUT
               MENUITEM "&Copy",             IDM_COPY
               MENUITEM "&Paste",            IDM_PASTE
               MENUITEM "De&lete",           IDM_DEL
               }
          POPUP "&Background"
               {
               MENUITEM "&White",            IDM_WHITE, CHECKED
               MENUITEM "&Lt Gray",          IDM_LTGRAY
               MENUITEM "&Gray",             IDM_GRAY
               MENUITEM "&Dk Gray",          IDM_DKGRAY
               MENUITEM "&Black",            IDM_BLACK
               }
          POPUP "&Help"
               {
               MENUITEM "&Help...",          IDM_HELP
               MENUITEM "&About PopMenu...", IDM_ABOUT
               }
          }
     }
```

POPMENU.H

```
/*------------------------
   POPMENU.H header file
  ------------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4
#define IDM_EXIT     5
```

```
#define IDM_UNDO    10
#define IDM_CUT     11
#define IDM_COPY    12
#define IDM_PASTE   13
#define IDM_DEL     14

#define IDM_WHITE   20
#define IDM_LTGRAY  21
#define IDM_GRAY    22
#define IDM_DKGRAY  23
#define IDM_BLACK   24

#define IDM_HELP    30
#define IDM_ABOUT   31
```

图 10-3 POPMENU程序

资源描述文件 POPMENU.RC 中定义的菜单同 MENUDEMO.RC 中的菜单非常相似，不同的是，在顶层菜单中只包含一项——一个弹出式菜单，它调用 File 、 Edit 、 Background 和 Help 选项。

在 WndProc 中的 WM_CREATE 处理期间， POPMENU 获取此弹出式菜单的句柄：

```
hMenu = LoadMenu (hInst, szAppName) ;
hMenu = GetSubMenu (hMenu, 0) ;
```

在处理 WM_RBUTTONDOWN 消息期间， POPMENU 提供了鼠标指针的位置，将此位置转换为屏幕坐标，再将坐标值传递给 TrackPopMenu :

```
point.x = LOWORD (lParam) ;
point.y = HIWORD (lParam) ;
ClientToScreen (hwnd, &point) ;

TrackPopupMenu (hMenu, 0, point.x, point.y, 0, hwnd, NULL) ;
```

然后 Windows 显示出具有 File 、 Edit 、 Background 和 Help 项的弹出式菜单，选择其中任何一项都可以使嵌套的弹出式菜单显示在右边，菜单函数同通常的菜单一样。

10.1.10 使用系统菜单

使用 WM_SYSMENU 风格创建的父窗口在其标题条的左侧有一个系统菜单框。如果你愿意，可以修改这个菜单。例如，你可以把自己的菜单命令加到系统菜单上。虽然不鼓励这种做法，但是修改系统菜单往往是一种快而不当的在短程序中添加菜单的方法。这里唯一的限制是：在系统菜单中增加的命令的 ID 值必须小于 F000H ；否则它们将会同 Windows 系统菜单命令所使用的 ID 值相冲突。还要记住：当你为这些新菜单项在窗口过程中处理 WM_SYSCOMMAND 消息时，你必须把其它的 WM_SYSCOMMAND 消息发送给 DefWindowProc 。如果你不这样做，那么实际上是禁止了系统菜单上的所有常规选项。

图 10-4 中所示的 POORMENU （“不当的个人菜单”）在系统菜单中加入了一个分隔条和三个命令，最后一个命令将删除这些附加的菜单项。

POORMENU.MAK

```
#------------------------
# POORMENU.MAK make file
#------------------------

poormenu.exe : poormenu.obj
     $(LINKER) $(GUIFLAGS) -OUT:poormenu.exe poormenu.obj $(GUILIBS)

poormenu.obj : poormenu.c
     $(CC) $(CFLAGS) poormenu.c
```

POORMENU.C

```
/*-----------------------------------------
   POORMENU.C -- The Poor Person's Menu
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>

#define IDM_ABOUT   1
#define IDM_HELP    2
#define IDM_REMOVE  3

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

static char szAppName[] = "PoorMenu" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HMENU      hMenu ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "The Poor-Person's Menu",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;

     hMenu = GetSystemMenu (hwnd, FALSE) ;

     AppendMenu (hMenu, MF_SEPARATOR, 0,           NULL) ;
     AppendMenu (hMenu, MF_STRING,    IDM_ABOUT,   "About...") ;
     AppendMenu (hMenu, MF_STRING,    IDM_HELP,    "Help...") ;
     AppendMenu (hMenu, MF_STRING,    IDM_REMOVE,  "Remove Additions") ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_SYSCOMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_ABOUT :
                         MessageBox (hwnd, "A Poor-Person's Menu Program.",
                                     szAppName, MB_OK | MB_ICONINFORMATION) ;
                         return 0 ;

                    case IDM_HELP :
                         MessageBox (hwnd, "Help not yet implemented!",
                                     szAppName, MB_OK | MB_ICONEXCLAMATION) ;
                         return 0 ;

                    case IDM_REMOVE :
                         GetSystemMenu (hwnd, TRUE) ;
                         return 0 ;
                    }
               break ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 10-4 POORMENU程序

三个菜单 ID 在 POORMENU.C 的开始部分定义：

```
#define IDM_ABOUT    1
#define IDM_HELP     2
#define IDM_REMOVE   3
```

在程序窗口创建之后，POORMENU 得到一个系统菜单的句柄：

```
hMenu = GetSystemMenu (hwnd, FALSE) ;
```

第一次调用 GetSystemMenu 时，你应该为修改菜单作准备，将第二个参数设置为 FALSE。

使用四个 AppendMenu 调用来实现对菜单的修改：

```
AppendMenu (hMenu, MF_SETAPATOR, 0,          NULL) ;
AppendMenu (hMenu, MF_STRING,    IDM_ABOUT,  "About...") ;
AppendMenu (hMenu, MF_STRING,    IDM_HELP,   "HELP...") ;
AppendMenu (hMenu, MF_STRING,    IDM_REMOVE, "Remove Additions") ;
```

第一个 AppendMenu 调用是添加分隔条。选择 Remove Additions 菜单项将使得 POORMENU 删除这些附加的菜单项，这只要把第二个参数设置为 TRUE，再次调用 GetSystemMenu 即可：

```
GetSystemMenu (hwnd, TRUE) ;
```

标准系统菜单有下列选项：Restore、Move、Size、Minimize、Close 和 Switch To。它们产生 wParam 分别等于 SC_RESTORE、SC_MOVE, SC_SIZE、SC_MINIMIZE、SC_CLOSE 和 SC_TASKLIST 的 WM_SYSCOMMAND 消息。尽管 Windows 程序一般不这样做，但是你可以自己处理这些消息，而不把它们留给 DefWindowProc。你也可以使用下面所述的方法来禁止或者除掉系统菜单的标准选项。Windows 文档中还介绍了一些系统菜单的标准附加项，这些附加项使用标识符 SC_NEXTWINDOW、SC_PREWINDOW、SC_VSCROLL、SC_HSCROLL 和 SC_ARRANGE。你也许会发现在一些应用程序中将这些命令加入系统菜单是合适的。

10.1.11 改变菜单

我们已经看到了如何使用 AppendMenu 函数为程序定义菜单以及将菜单项加入到系统菜单中。在 Windows 3.0 之前，你不得不被迫使用 ChangeMenu 函数来完成这种工作。ChangeMenu 函数功能很多，在 Windows 的所有版本中它是最复杂的函数之一。在 Windows 95 中，ChangeMneu 函数仍然是可用的，不过，它的功能被分解为五个新的函数：

- AppendMenu——在菜单尾部添加一个新的菜单项
- DeleteMenu——删除菜单中一个现有的菜单项并清除该项
- InsertMenu——在菜单中插入一个新项
- ModifyMenu——修改一个现有的菜单项
- RemoveMenu——从菜单中移走某一项

如果菜单项是一个弹出式菜单，那么 DeleteMenu 和 RemoveMenu 之间的区别就很重

要， DeleteMenu 清除弹出式菜单；但 RemoveMenu 却并不清除它。

10.1.12　其它菜单命令

下面是在使用菜单时一些有用的函数。

当你改变顶层菜单项时，直到 Windows 重画菜单条时才显示所作的改变，你可以通过下列调用来强迫执行菜单刷新：

```
DrawMenuBar (hwnd) ;
```

应该注意的是， DrawMenuBar 的参数是窗口句柄而不是菜单句柄。

你可以使用下列命令来获得弹出式菜单的句柄：

```
hMenuPopup = GetSubMenu (hMenu, iPosition) ;
```

其中 iPosition 是 hMenu 指示的顶层菜单中弹出式菜单项的索引（开始为 0 ）。然后你可以在其它函数中使用弹出式菜单句柄（例如在 AppendMenu 函数中）。

你可以使用下列命令获得顶层菜单或者弹出式菜单中的当前项数：

```
iCount = GetMenuItemCount (hMenu) ;
```

你可以获取弹出式菜单项的菜单 ID：

```
id = GetMenuItemID (hMenuPopup, iPosition) ;
```

其中 iPosition 是菜单项在弹出式菜单中的位置（以 0 开始）。

在 MENUDEMO 中你已经看到如何选中或者取消选中弹出式菜单中的某一项：

```
CheckMenuItem (hMenu, id, iCheck) ;
```

在 MENUDEMO 中， hMenu 是顶层菜单的句柄， id 是菜单 ID ，而 iCheck 的值是 MF_CHECKED 或者是 MF_UNCHECKED 。如果 hMenu 是弹出式菜单句柄，那么参数 id 是位置索引而不是菜单 ID 。如果使用索引更方便的话，那么你可以在第三个参数中包含 MF_BYPOSITION，例如：

```
CheckMenuItem (hMenu, iPosition, MF_CHECKED | MF_BYPOSITION) ;
```

除了第三个参数是 MF_ENABLED 、 MF_DISABLED 或者 MF_GRAYED 外， EnableMenuItem 函数与 CheckMenuItem 函数所完成的工作类似。如果你在具有弹出式菜单的顶层菜单项上使用 EnableMenuItem ，那么必须在第三个参数中使用 MF_BYPOSITION 标识符，因为菜单项没有菜单 ID 。我们将在本章后面所示的 POPPAD 程序中看到 EnableMenuItem 的一个例子。 HiliteMenuItem 也类似于 CheckMenuItem 和 EnableMenuItem ，但是它使用的是 MF_HILITE 和 MF_UNHILITE 。当你在菜单项之间移动时， WIndows 使用反显方式加亮显示菜单项。你通常不需要使用 HiliteMenuItem 。

你还需要对你的菜单做些什么呢？你还记得我们在菜单中使用了一些什么字符串吗？你可以通过下面的调用来回顾一下：

```
iByteCount = GetMenuString (hMenu, id, pString, iMaxCount, iFlag) ;
```

iFlag 可以是 MF_BYCOMMAND（其中 id 是菜单 ID），也可以是 MF_BYPOSITION（其中 id 是位置索引）。函数将字符串的 iMaxCount 个字节复制到 pString 中，并返回拷贝的字节数。

或者你也许想知道菜单项的当前属性是什么：

```
iFlags = GetMenuState (hMenu, id, iFlag) ;
```

同样地，iFlag 可以是 MF_BYCOMMAND 或者是 MF_BYPOSITION 。返回值 iFlags 是当前所有属性的组合，你可以通过对 MF_DISABLED 、 MF_GRAYED 、 MF_MENUBREAK 、 MF_MENUBARBREAK 和 MF_SEPARATOR 标识符的检测来决定当前的属性。

也许现在你对菜单有了一些了解。这时你可能想知道如果你不再需要菜单时又应该如何处理。你可以使用下面的命令来清除菜单：

```
DestroyMenu (hMenu) ;
```

从而使菜单句柄无效。

10.1.13 创建菜单的非正统方法

现在让我们稍微偏离我们所讨论的主题。如果在你的程序中没有下拉式菜单，而是创建了多个没有弹出式菜单的顶层菜单，并调用 SetMenu 在顶层菜单之间切换会是什么样的情形呢？图 10-5 中的 NOPOPUPS 程序演示了如何处理这种情况。在这个程序中，弹出式菜单项 Edit 和 File 与 MENUDEMO 程序中的类似，但是却作为另一种顶层菜单显示出来。

NOPOPUPS.MAK

```
#-----------------------
# NOPOPUPS.MAK make file
#-----------------------

nopopups.exe : nopopups.obj nopopups.res
     $(LINKER) $(GUIFLAGS) -OUT:nopopups.exe nopopups.obj \
     nopopups.res $(GUILIBS)

nopopups.obj : nopopups.c nopopups.h
     $(CC) $(CFLAGS) nopopups.c

nopopups.res : nopopups.rc nopopups.h
     $(RC) $(RCVARS) nopopups.rc
```

NOPOPUPS.C

```
/*-------------------------------------------------
   NOPOPUPS.C -- Demonstrates No-Popup Nested Menu
                 (c) Charles Petzold, 1996
  -------------------------------------------------*/

#include <windows.h>
#include "nopopups.h"
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "NoPopUps" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "No-Popup Nested Menu Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HMENU hMenuMain, hMenuEdit, hMenuFile ;
     HINSTANCE    hInstance ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

               hMenuMain = LoadMenu (hInstance, "MenuMain") ;
               hMenuFile = LoadMenu (hInstance, "MenuFile") ;
               hMenuEdit = LoadMenu (hInstance, "MenuEdit") ;
```

```
               SetMenu (hwnd, hMenuMain) ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_MAIN :
                         SetMenu (hwnd, hMenuMain) ;
                         return 0 ;

                    case IDM_FILE :
                         SetMenu (hwnd, hMenuFile) ;
                         return 0 ;

                    case IDM_EDIT :
                         SetMenu (hwnd, hMenuEdit) ;
                         return 0 ;

                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep (0) ;
                         return 0 ;
                    }
               break ;

          case WM_DESTROY :
               SetMenu (hwnd, hMenuMain) ;
               DestroyMenu (hMenuFile) ;
               DestroyMenu (hMenuEdit) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

NOPOPUPS.RC

```
/*-----------------------------
   NOPOPUPS.RC resource script
  -----------------------------*/

#include "nopopups.h"

MenuMain MENU
     {
     MENUITEM "MAIN:",          0,          INACTIVE
```

```
     MENUITEM "&File...",       IDM_FILE
     MENUITEM "&Edit...",       IDM_EDIT
     }

MenuFile MENU
     {
     MENUITEM "FILE:",          0,          INACTIVE
     MENUITEM "&New",           IDM_NEW
     MENUITEM "&Open...",       IDM_OPEN
     MENUITEM "&Save",          IDM_SAVE
     MENUITEM "Save &As...",    IDM_SAVEAS
     MENUITEM "(&Main)",        IDM_MAIN
     }

MenuEdit MENU
     {
     MENUITEM "EDIT:",          0,          INACTIVE
     MENUITEM "&Undo",          IDM_UNDO
     MENUITEM "Cu&t",           IDM_CUT
     MENUITEM "&Copy",          IDM_COPY
     MENUITEM "&Paste",         IDM_PASTE
     MENUITEM "De&lete",        IDM_DEL
     MENUITEM "(&Main)",        IDM_MAIN
     }
```

NOPOPUPS.H

```
/*-------------------------
   NOPOPUPS.H header file
  -------------------------*/

#define IDM_NEW     1
#define IDM_OPEN    2
#define IDM_SAVE    3
#define IDM_SAVEAS  4

#define IDM_UNDO    5
#define IDM_CUT     6
#define IDM_COPY    7
#define IDM_PASTE   8
#define IDM_DEL     9

#define IDM_MAIN    10
#define IDM_EDIT    11
#define IDM_FILE    12
```

图 10-5 NOPOPUPS程序

资源描述文件中有三个菜单，而不是一个。当窗口过程处理 WM_CREATE 消息时，Windows 把每个菜单资源都装入内存：

```
hMenuMain = LoadMenu (hInstance, "MenuMain") ;
hMenuFile = LoadMenu (hInstance, "MenuFile") ;
hMenuEdit = LoadMenu (hInstance, "MenuEdit") ;
```

开始时，程序只显示主菜单：

```
SetMenu (hwnd, hMenuMain) ;
```

主菜单使用字符串“ MAIN:”、“ File...”和“ Edit...”列出这三个选项。然而，“ MAIN:”是禁用的，因此它不能使 WM_COMMAND 消息被发送到窗口过程。 File 和 Edit 菜单项以“ File:”和“ Edit:”开始，表明它们是子菜单。每个菜单的最后一项都是字符串“(&Main)”，表示返回到主菜单，以便在这三个菜单之间进行切换：

```
case WM_COMMAND :
     switch (LOWORD (wParam))
          {
          case IDM_MAIN :
               SetMenu (hwnd, hMenuMain) ;
               return 0 ;

          case IDM_FILE :
               SetMenu (hwnd, hMenuFile) ;
               return 0 ;

          case IDM_EDIT :
               SetMenu (hwnd, hMenuEdit) ;
               return 0 ;

          [other program lines]
          }
     break ;
```

在处理 WM_DESTROY 消息期间， NOPOPUPS 将程序的菜单设置为主菜单，并调用 DestroyMenu 来清除 File 和 Edit 菜单。当窗口被清除时，主菜单将被自动清除。

10.2 在菜单中使用位图

字符串并不是显示菜单项的唯一方式，你也可以使用位图来显示菜单。如果你立即想到的是菜单的文件夹、胶水瓶和垃圾箱等图象，那么请不要考虑这些图象；而是想一想对画图程序来说，菜单位图是多么有用，想一想如何在菜单中使用不同的字体和字体大小、行宽、条纹图案以及颜色等等。

我们将要剖析的是一个叫做 GRAFMENU（“图形菜单”）的程序，顶层菜单如图 10-6 中所示。放大的字符块是由 Developer Studio 的图象编辑器产生的 40x16 象素单色位图文件得来的，并且被存放在扩展名为.BMP 的文件中；它们也可以是图片。从菜单中选择 FONT 激活一个包含三个选项—— Courier New 、 Arial 和 Times New Roman ——的弹出式菜单，这些都是 TrueType 字体，并且每一种都按各自的字体显示（图 10-7 ）。在程序中产生位图时，使用了一种涉及内存设备环境的技术。

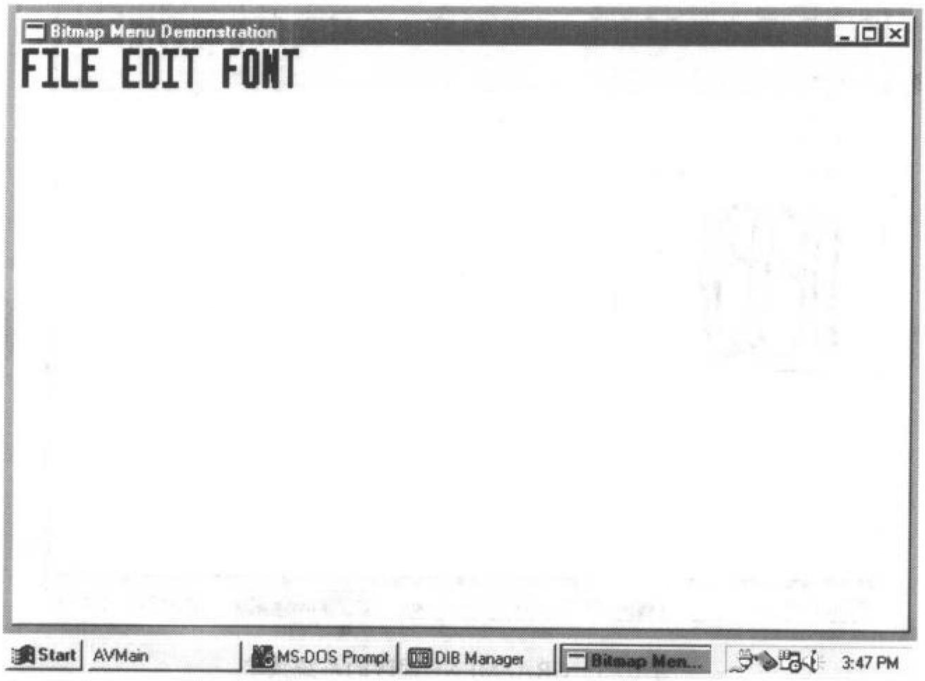


图 10-6　GRAFMENU程序的顶层菜单

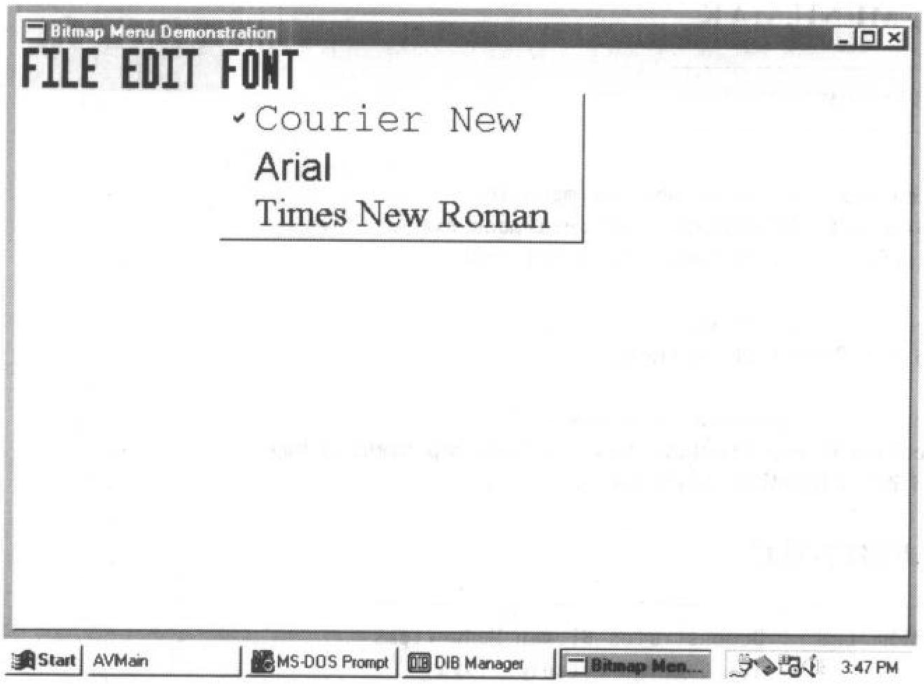


图 10-7　GRAFMENU程序的FONT弹出式菜单

最后，当你下拉系统菜单时，会发现可以得到一些“帮助”信息，里面的单词“ Help ”似乎正表示了一个新用户的绝望（图 10-8 ）。这个 64x64 象素的单色位图是在 Developer Studio 的图象编辑器中创建的。

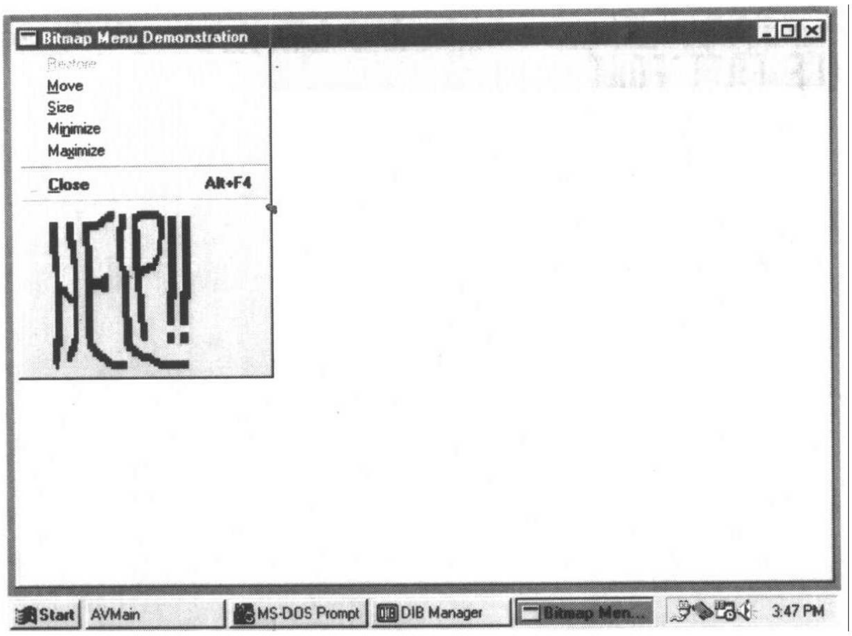


图 10-8 GRAFMENU程序的系统菜单

GRAFMENU 程序包含了在图象编辑器中创建的位图，如图 10-9 所示。

GRAFMENU.MAK

```
#------------------------
# GRAFMENU.MAK make file
#------------------------

grafmenu.exe : grafmenu.obj grafmenu.res
     $(LINKER) $(GUIFLAGS) -OUT:grafmenu.exe \
     grafmenu.obj grafmenu.res $(GUILIBS)

grafmenu.obj : grafmenu.c grafmenu.h
     $(CC) $(CFLAGS) grafmenu.c

grafmenu.res : grafmenu.rc grafmenu.h \
     editlabl.bmp filelabl.bmp fontlabl.bmp bighelp.bmp
     $(RC) $(RCVARS) grafmenu.rc
```

GRAFMENU.C

```
/*----------------------------------------------
   GRAFMENU.C -- Demonstrates Bitmap Menu Items
                 (c) Charles Petzold, 1996
  ----------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "grafmenu.h"

LRESULT CALLBACK WndProc  (HWND, UINT, WPARAM, LPARAM) ;
```

```
HBITMAP StretchBitmap (HBITMAP) ;
HBITMAP GetBitmapFont (int) ;

char szAppName[] = "GrafMenu" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HBITMAP    hBitmapHelp, hBitmapFile, hBitmapEdit,
                hBitmapFont, hBitmapPopFont[3] ;
     HMENU      hMenu, hMenuPopup ;
     HWND       hwnd ;
     int        i ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hMenu = CreateMenu () ;

     hMenuPopup = LoadMenu (hInstance, "MenuFile") ;
     hBitmapFile = StretchBitmap (LoadBitmap (hInstance, "BitmapFile")) ;
     AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                 (PSTR) (LONG) hBitmapFile) ;

     hMenuPopup = LoadMenu (hInstance, "MenuEdit") ;
     hBitmapEdit = StretchBitmap (LoadBitmap (hInstance, "BitmapEdit")) ;
     AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                 (PSTR) (LONG) hBitmapEdit) ;

     hMenuPopup = CreateMenu () ;

     for (i = 0 ; i < 3 ; i++)
          {
          hBitmapPopFont[i] = GetBitmapFont (i) ;
          AppendMenu (hMenuPopup, MF_BITMAP, IDM_COUR + i,
                      (PSTR) (LONG) hBitmapPopFont[i]) ;
          }

     hBitmapFont = StretchBitmap (LoadBitmap (hInstance, "BitmapFont")) ;
     AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int) hMenuPopup,
                 (PSTR) (LONG) hBitmapFont) ;
```

```
     hwnd = CreateWindow (szAppName, "Bitmap Menu Demonstration",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, hMenu, hInstance, NULL) ;

     hMenu = GetSystemMenu (hwnd, FALSE);
     hBitmapHelp = StretchBitmap (LoadBitmap (hInstance, "BitmapHelp")) ;
     AppendMenu (hMenu, MF_SEPARATOR, NULL,     NULL) ;
     AppendMenu (hMenu, MF_BITMAP,    IDM_HELP, (PSTR) (LONG) hBitmapHelp) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     DeleteObject (hBitmapHelp) ;
     DeleteObject (hBitmapEdit) ;
     DeleteObject (hBitmapFile) ;
     DeleteObject (hBitmapFont) ;

     for (i = 0 ; i < 3 ; i++)
          DeleteObject (hBitmapPopFont[i]) ;

     return msg.wParam ;
     }

HBITMAP StretchBitmap (HBITMAP hBitmap1)
     {
     BITMAP     bm1, bm2 ;
     HBITMAP    hBitmap2 ;
     HDC        hdc, hdcMem1, hdcMem2 ;
     TEXTMETRIC tm ;

     hdc = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
     GetTextMetrics (hdc, &tm) ;
     hdcMem1 = CreateCompatibleDC (hdc) ;
     hdcMem2 = CreateCompatibleDC (hdc) ;
     DeleteDC (hdc) ;

     GetObject (hBitmap1, sizeof (BITMAP), (PSTR) &bm1) ;

     bm2 = bm1 ;
     bm2.bmWidth      = (tm.tmAveCharWidth * bm2.bmWidth)  / 4 ;
     bm2.bmHeight     = (tm.tmHeight       * bm2.bmHeight) / 8 ;
     bm2.bmWidthBytes = ((bm2.bmWidth + 15) / 16) * 2 ;

     hBitmap2 = CreateBitmapIndirect (&bm2) ;

     SelectObject (hdcMem1, hBitmap1) ;
```

```
     SelectObject (hdcMem2, hBitmap2) ;

     StretchBlt (hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
                 hdcMem1, 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY) ;

     DeleteDC (hdcMem1) ;
     DeleteDC (hdcMem2) ;
     DeleteObject (hBitmap1) ;

     return hBitmap2 ;
     }

HBITMAP GetBitmapFont (int i)
     {
     static char   *szFaceName[3] = { "Courier New", "Arial",
                                      "Times New Roman" } ;
     static LOGFONT lf ;
     HBITMAP        hBitmap ;
     HDC            hdc, hdcMem ;
     HFONT          hFont ;
     SIZE           size ;
     TEXTMETRIC     tm ;

     hdc = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
     GetTextMetrics (hdc, &tm) ;

     lf.lfHeight = 2 * tm.tmHeight ;
     strcpy ((char *) lf.lfFaceName, szFaceName[i]) ;

     hdcMem = CreateCompatibleDC (hdc) ;
     hFont = (HFONT) SelectObject (hdcMem, CreateFontIndirect (&lf)) ;
     GetTextExtentPoint (hdcMem, szFaceName[i],
                         strlen (szFaceName[i]), &size) ;

     hBitmap = CreateBitmap (size.cx, size.cy, 1, 1, NULL) ;
     SelectObject (hdcMem, hBitmap) ;

     TextOut (hdcMem, 0, 0, szFaceName[i], strlen (szFaceName[i])) ;

     DeleteObject (SelectObject (hdcMem, hFont)) ;
     DeleteDC (hdcMem) ;
     DeleteDC (hdc) ;

     return hBitmap ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HMENU      hMenu ;
     static int iCurrentFont = IDM_COUR ;

     switch (iMsg)
          {
          case WM_CREATE :
               CheckMenuItem (GetMenu (hwnd), iCurrentFont, MF_CHECKED) ;
```

```
                return 0 ;

          case WM_SYSCOMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_HELP :
                         MessageBox (hwnd, "Help not yet implemented!",
                                     szAppName, MB_OK | MB_ICONEXCLAMATION) ;
                         return 0 ;
                    }
               break ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_UNDO :
                    case IDM_CUT :
                    case IDM_COPY :
                    case IDM_PASTE :
                    case IDM_DEL :
                         MessageBeep (0) ;
                         return 0 ;

                    case IDM_COUR :
                    case IDM_ARIAL :
                    case IDM_TIMES :
                         hMenu = GetMenu (hwnd) ;
                         CheckMenuItem (hMenu, iCurrentFont, MF_UNCHECKED) ;
                         iCurrentFont = LOWORD (wParam) ;
                         CheckMenuItem (hMenu, iCurrentFont, MF_CHECKED) ;
                         return 0 ;
                    }
               break ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

GRAFMENU.RC

```
/*-----------------------------
   GRAFMENU.RC resource script
  -----------------------------*/

#include "grafmenu.h"

BitmapEdit BITMAP editlabl.bmp
BitmapFile BITMAP filelabl.bmp
```

```
BitmapFont BITMAP fontlabl.bmp
BitmapHelp BITMAP bighelp.bmp

MenuFile MENU
     {
     MENUITEM "&New",          IDM_NEW
     MENUITEM "&Open...",      IDM_OPEN
     MENUITEM "&Save",         IDM_SAVE
     MENUITEM "Save &As...",   IDM_SAVEAS
     }
MenuEdit MENU
     {
     MENUITEM "&Undo",         IDM_UNDO
     MENUITEM SEPARATOR
     MENUITEM "Cu&t",          IDM_CUT
     MENUITEM "&Copy",         IDM_COPY
     MENUITEM "&Paste",        IDM_PASTE
     MENUITEM "De&lete",       IDM_DEL
     }
```

GRAFMENU.H

```
/*------------------------
   GRAFMENU.H header file
  ------------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4

#define IDM_UNDO     5
#define IDM_CUT      6
#define IDM_COPY     7
#define IDM_PASTE    8
#define IDM_DEL      9

#define IDM_COUR     10
#define IDM_ARIAL    11
#define IDM_TIMES    12

#define IDM_HELP     13
```

EDITLABL.BMP

FILELABL.BMP

FONTLABL.BMP

BIGHELP.BMP

图 10-9 GRAFMENU程序

10.2.1 创建菜单位图的两种方法

在菜单中插入位图，你可以使用 AppendMenu 和 InsertMenu 。而位图是从哪里来的呢？它来自于下面两个地方之一。第一，你可以使用 Developer Studio 的图象编辑器来创建一个位图，并且将位图文件包含在你的资源描述文件中。在程序中，你可以使用 LoadBitmap 将位图资源装入内存，并且使用 AppendMenu 或者 InsertMenu 将它插入菜单。然而，使用这种方法有一个问题。这种方法并不适合于所有不同的视频分辨率以及纵横比的显示器；你必须减小或者放大所装入的位图来解决这个问题。另一种方法是在程序中创建位图并把它插入菜单。

这两种方法看起来都比实际上要困难得多。我们不必为其本身浪费时间。 Windows 提供了一些函数，以便我们利用“内存设备环境”技术干净利落地处理位图。

10.2.2 内存设备环境

当你使用 GDI 调用（诸如 TextOut 等）在窗口的客户区中写文本时，你实际上是在向一块内存（视频显示内存）中写文本，这块内存组织得非常像一个巨型的位图，其宽和高等于视频显示器的分辨率，多个位定义颜色的方式也是由视频适配器定义的。如果你想一想， Windows 还应该能够将一块普通的内存视为视频显示内存，能够像在屏幕上写东西一样在这块内存上写数据。这样，我们应该能够将这块内存用作一个位图。

这正是内存设备环境，它有助于我们绘制和处理 Windows 程序中的位图。下面是所涉及的步骤：

1. 使用 CreateCompatibleDC 调用创建内存设备环境。开始时，内存设备环境的显示表面是一个单色象素，你可以认为这个设备环境是双色的（黑和白）， 1 个象素高， 1 个象素宽。
2. 使用 CreateBitmap 、 CreateBitmapIndirect 或 CreateCompatibleBitmap 创建一个未

初始化的位图。当你创建位图时，指定高度和宽度以及颜色组织。然而，位图的象素实际尚未表示任何东西。保存此位图的句柄。

3. 使用 SelectObject 将位图选进内存设备环境。现在内存设备环境中有一个与位图具有同样大小、与位图定义的彩色平面个数相同的显示平面。
4. 使用 GDI 函数在内存设备环境中画图，其方法与使用 GDI 函数在一般的设备环境中画图的方法相同。你在内存设备环境的显示表面上绘制的所有内容实际上被送入选进设备环境的位图上。
5. 删除内存设备环境。你现在还有位图句柄，位图中包含着你在内存设备环境中所绘制内容的象素表示。

10.2.3　用文本创建位图

GRAFMENU 中的 GetBitmapFont 函数接受参数 0 、 1 或者 2，并且返回一个位图句柄。该位图以适当的字体包含字符串 “ Courier New ” 、 “ Arial ” 或 “ Times New Roman ”，这些字体的尺寸大约是系统正常字体的两倍。让我们来看一看 GetBitmapFont 是如何实现的。（下面的代码与 GRAFMENU.C 文件中的代码不同，为清晰起见，我用适合于 Arial 的值来取代对 szFaceName 数组的引用。）

第一步是通过使用 TEXTMETRIC 结构确定系统字体的大小：

```
hdc = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
GetTextMetrics (hdc, &tm) ;
```

这个 TEXTMETRIC 结构的某些域必须被修改，使它能够在一个逻辑字体结构中描述更大的 Arial 字体：

```
lf.lfHeight = 2*tm.tmHeight;
strcpy ( (char*)lf.lfFaceName, "Arial"):
```

下一步是获取屏幕的设备环境，并且创建一个与屏幕兼容的内存设备环境：

```
hdcMem = CreateCompatibleDC (hdc) ;
```

内存设备环境句柄是 hdcMem 。然后，我们在修改过的 lf 结构基础上创建一种字体，并将它选入内存设备环境中：

```
hFont = (HFONT)SelectObject (hdcMem, CreateFontIndirect (&lf)) ;
```

现在，当我们向内存设备环境中写入文本时， Windows 将使用选入设备环境中的 True Type Arial 字体。

但是这个内存设备环境仍然只有一个单象素的单色设备表面，我们必须创建一个足够大的位图，以便在单色显示表面上显示文本。你可以通过 GetTextExtentPoint 来获取文本的尺寸，并使用 CreateBitmap 创建一个基于这个尺寸的位图：

```
GetTextExtentPoint (hdcMem, "Arial", 5, &size) ;
hBitmap = CreateBitmap (size.cx, size.cy, 1, 1, NULL) ;
SelectObject (hdcMem, hBitmp) ;
```

这个设备环境的单色显示表面现在正好是文本的大小，我们需要做的一切就是将文本写到显示表面上。你以前已经看到过这个函数：

```
TextOut (hdcMem, 0, 0, "Arial", 5) ;
```

这样我们就完成了写操作，只剩下结束处理了。为此，我们使用 SelectObject 将系统字体（句柄为 hFont ）选回设备环境中，并删除先前 SelectObject 返回的字体句柄，这是 Arial 字体的句柄：

```
DeleteObject (SelectObject (hdcMenu, hFont)) ;
```

现在我们还必须删除两个设备环境：

```
DeleteDC (hdcMem) ;
DeleteDC (hdc) ;
```

这样我们就只留下了包含 Arial 字体的“ Arial ”文本位图了。

10.2.4 位图的比例缩放

当我们需要对字体进行缩放，以便适应不同的显示分辨率和纵横比时，内存设备环境起着很重要的作用。我在 GRAFMENU 中创建了四种位图，它们适合于系统字体为 8 个象素宽， 4 个象素高的显示器。对于其它尺寸的系统字体，需要对位图进行放大，这项工作由 GRAFMENU 程序中的 StretchBitmap 函数来实现。

首先要做的是获取屏幕的设备环境，获取系统字体的文本尺寸和创建两个内存设备环境：

```
hdc = CreateIC ("DISPLAY", NULL, NULL, NULL) ;
GetTextMetrics (hdc, &tm) ;
hdcMenu1 = CreateCompatibleDC (hdc) ;
hdcMenu2 = CreateCompatibleDC (hdc) ;
DeleteDC (hdc) ;
```

传递到函数的位图句柄是 hBitmap1 ，程序可以通过使用 GetObject 获取位图的尺寸：

```
GetObject (hBitmap1, sizeof (BITMAP), (PSTR)&bm1) ;
```

从而将尺寸复制到 BITMAP 类型的结构 bm1 中。结构 bm2 被置为与 bm1 相等，然后以系统字体尺寸为依据修改某些域：

```
bm2 = bm1;
bm2.bmWidth = (tm.tmAveCharWidth * bm2.bmWidth) / 4;
bm2.bmHeight = (tm.tmHeigth * bm2.bmHeight) / 8;
bm.bmWidthBytes = ((bm2.bmWidth + 15) / 16) * 2;
```

然后可以创建具有句柄 hBitmap2 的新位图，它是基于改变后的尺寸的：

```
hBitmap2 = CreateBitmapIndirect (&bm2) ;
```

接着你可以将这两个位图选入到两个内存设备环境中：

```
SelectObject (hdcMem1, hBitmap1) ;
```

```
SelectObject (hdcMem2, hBitmap2) ;
```

我们要把第一个位图拷贝到第二个位图中去，并且在处理过程中缩放第二个位图。这里涉及到 StretchBlt 调用：

```
StretchBlt (hdcMem2, 0, 0, bm2.bmWidth, bm2.bmHeight,
            hdcMem1 0, 0, bm1.bmWidth, bm1.bmHeight, SRCCOPY) ;
```

现在，第二个位图是经过适当变换后的位图，我们可以在菜单中使用它。结束工作很简单：

```
DeleteDC (hdcMem1) ;
DeleteDC (hdcMem2) ;
DeleteObject (hBitmap1) ;
```

10.2.5 将位图放入菜单中

在构造菜单时， GRAFMENU 程序中的 WinMain 函数使用了 StretchBitmap 和 GetBitmapFont 这两个函数。 GRAFMENU 在资源描述文件中已经定义了两个菜单，它们将成为 File 和 Edit 选项的弹出式菜单。

GRAFMENU 从获得一个空的菜单句柄开始：

```
hMenu = CreateMenu () ;
```

弹出式菜单 File（包括四个选项 New 、 Open 、 Save 和 Svae As ）被从资源描述文件中装入内存：

```
hMenuPopup = LoadMenu (hInstance, "MenuFile") ;
```

包含串“ File ”的位图也被从资源描述文件中装入，并且使用 StretchBitmap 来放大它：

```
hBitmapFile = StretchBitmap (LoadBitmap (hInstance, "BitmapFile") ;
```

位图句柄和弹出式菜单句柄被作为参数来调用 AppendMenu ：

```
AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int)hMenuPopup,
           (PSTR) (LONG)hBitmapFile) ;
```

Edit 菜单遵照同样的过程：

```
hMenuPopup = LoadMenu (hInstance, "MenuEdit") ;
hBitmapEdit = StretchBitmap (LoadBitmap (hInstance, "BitmapEdit")) ;
AppendMenu (hMenu, MF_BITMAP | MF_POPUP, (int)hMenuPopup,
           (PSTR) (LONG)hBitmapEdit) ;
```

弹出式菜单中的三种字体是通过调用 GetBitmapFont 函数来构造的：

```
hMenuPopup = CreateMenu () ;
for  (I=0; I<3; I++)
    {
    hBitmapPopFont[i] = GetBitmapFont (i) ;
```

```
    AppendMenu (hMenuPopup, MF_BITMAP, IDM_COUR + I,
                (PSTR) (LONG)hMenuPopupFont[i]) ;
    }
```

然后弹出式菜单被添加到菜单中：

```
hBitmapFont = StretchBitmap (LoadBitmap (hInstance, "BitmapFont") ;
AppendMenu (hMenu, MF_BITMAP | MF_POPUP,  (int)hMenuPopup,
            (PSTR) (LONG)hBitmapFont) ;
```

到此为止，窗口菜单已经构造完毕。现在你可以在 CreateWindow 调用中包含 hMenu 了：

```
hwnd = CreateWindow (szAppName, "Bitmap Menu Demonstration",
                     WS_OVERLAPPEDWINDOW,
                     CW_USEDEFAULT, CW_USEDEFULT,
                     CW_USEDEFAULT, CW_USEDEFULT,
                     NULL, hMenu, hInstance, NULL) ;
```

在得到 hwnd 之后， GRAFMENU 程序就可以改变系统菜单。 GRAFMENU 程序首先获取一个系统菜单的句柄：

```
hMenu = GetSystemMenu (hwnd, FALSE) ;
```

下列语句加载"Help"位图，并且把它放大到某种适当的尺寸：

```
hBitmap = StretchBitmap (LoadBitmap (hInstance, "BitmapHelp")) ;
```

在系统菜单中加入一个分隔条，并且放大系统菜单位图，命令如下：

```
AppendMenu (hMenu, MF_SEPARATOR, NULL, NULL) ;
AppendMenu (hMenu, MF_BITMP, IDM_HELP, (PSTR) (LONG)hBitmapHelp) ;
```

请记住，位图都是 GDI 对象，必须在程序结束之前删除它们。当 GRAFMENU 从它的消息循环中退出时，你应该使用下面的语句完成这项工作：

```
DeleteObject (hBitmapHelp) ;
DeleteObject (hBitmapEdit) ;
DeleteObject (hBitmapFile) ;
DeleteObject (hBitmapFont) ;

for (I=0; I<3; I++)
    DeleteObject (hBitmapPopFont[i]) ;
```

我以下面的一些注意事项结束此节：

- 在顶层菜单中， Windows 调整菜单条的高度以容纳最高的位图，其它位图（或者字符串）向菜单条的顶部对齐。菜单条的尺寸由下面这个函数调用来获取：

  ```
  GetSystemMetrics (SM_CYMENU) ;
  ```

  在把位图放入顶层菜单之后，这个尺寸就不再有效了。

- 从 GRAFMENU 的运行结果中可以看到，你能够将选中标记用于位图化的弹出式菜单项，但是选中标记是正常的尺寸。如果你觉得麻烦，那么可以创建一个自定

义的选中标记并且使用 SetMenuItemBitmaps 。

- 另一种使用非文本(或者非系统字体的文本)的方法是采用“自绘(owner-draw)”菜单。

10.2.6　增加键盘接口

现在我们遇到另一个问题。当菜单包含文本时， Windows 将自动加入键盘接口。你可以使用 Alt 键与字符串中的一个字母相结合来选择菜单项，但是一旦菜单中放入了位图，键盘接口就失效了。即使位图代表一定的操作， Windows 也不知道。

这正是 WM_MENUCHAR 消息的用途所在。当你按一个并不对应某一菜单项的字符键和 Alt 键时， Windows 发送 WM_MENUCHAR 消息给窗口过程。这样，我们需要截取 WM_MENUCHAR 消息，并检查 wParam 的低位值（按下的键所对应的 ASCII 字符）。如果该值与菜单项一致，那么我们必须返回一个长整数给 Windows ，这个长整数的高位字为 2 ，低位字是我们想让该键与之相关联的菜单项的索引值。其它工作由 Windows 完成。

10.3　加 速 键

简单说来，加速键是产生 WM_COMMAND 消息（有些情况下是 WM_SYSCOMMAND ）的键组合。许多时候，程序使用加速键来重复常用菜单项的动作（然而，加速键还可以用于执行非菜单功能）。例如，许多 Windows 程序都有一个包含 Delete 选项的 Edit 菜单，这些程序习惯上都将 Del 键指定为 Delete 选项的加速键。用户可以通过“ Alt-键”从菜单中选择 Delete 选项，或者只需按下加速键 Del 。当窗口过程收到一条 WM_COMMAND 消息时，它不必确定使用的是菜单还是加速键。

10.3.1　为什么要使用加速键

你也许会问：为什么我应该使用加速键？为什么不能直接捕获 WM_KEYDOWN 或者 WM_CHAR 消息而自己来实现同样的菜单功能呢？好处在哪里？对于一个单窗口应用程序，你当然可以捕获键盘消息，但是使用加速键可以得到一些好处：你不需要复制菜单和加速键逻辑。

对于有多个窗口和多个窗口过程的应用程序来说，加速键是非常重要的。正如我们所看到的， Windows 把 WM_COMMAND 消息发送给窗口过程，该窗口过程的句柄在 Windows 函数 TranslateAccelerater 中给出。通常这是主窗口，也是拥有菜单的窗口，这意味着无需每个窗口过程都复制加速键的操作逻辑。

如果你在主窗口的客户区中使用了非模态对话框（在下一章中讨论）或者子窗口，那么这种好处就变得非常重要。如果定义一个特定的加速键以便在不同的窗口之间移动，那么，只需要一个窗口过程有这个逻辑，子窗口不会收到加速键引发的 WM_COMMAND 消息。

10.3.2 安排加速键的几条规则

从理论上讲，你可以使用任何虚拟键或者字符键同 Shift 键、Ctrl 键或 Alt 键一起来定义加速键。然而，你应该尽力使得应用程序之间协调一致，并且尽量避免干扰 Windows 的键盘使用。在加速键中，应该避免使用 Tab 、 Enter 、 Esc 和 Spacebar 键，因为这些键常常用于完成系统功能。

加速键最经常的用途是操作 Edit 菜单中的各项。为这些菜单项推荐的加速键在 Windows 3.0 和 Windows 3.1 之间已有不同，因此通常都要支持如下所列的新旧两套加速键：

功能	旧加速键	新加速键
撤消	Alt+Backspace	Ctrl+Z
剪切	Shift+Del	Ctrl+X
复制	Ctrl+Ins	Ctrl+C
粘贴	Shift+Ins	Ctrl+V
删除或清除	Del	Del

另一种常用的虚拟键是激活帮助信息的功能键 F1 。应该避免使用 F4 、 F5 和 F6 键，因为这些键常用在多文档界面（ MDI ）程序中完成特殊的功能（将在第 18 章中讨论）。

10.3.3 加速键表

加速键定义在你的资源描述文件中，其一般形式如下所示：

```
MyAccelerators ACCELERATORS
     {
     [加速键定义]
     }
```

这个加速键表的名字为 MyAccelerators 。 ACCELERATORS 表不包含装入和内存选项。你可以在你的资源描述文件中包含多个 ACCELERATORS 表。

不同的加速键必须定义在表的不同行上。有四种类型的加速键定义：

```
"char",   id              [.SHIFT] [.CONTROL] [.ALT]
"^char",  id              [.SHIFT] [.CONTROL] [.ALT]
nCode,    id,  ASCII      [.SHIFT] [.CONTROL] [.ALT]
nCode,    id,  VIRKEY     [.SHIFT] [.CONTROL] [.ALT]
```

在这些例子中，“ char ”表示双引号内的单个字符，“^char ”表示双引号内的字符^和单个字符。 id 号的功能类似于菜单定义中的菜单 ID，它的值是 Windows 用 WM_COMMAND 选项发送给窗口过程以标识加速键的值，这些值通常是定义在头文件中的标识符。当加速

键复制菜单命令时，对菜单和加速键使用同一个 ID 。当加速键不复制菜单命令时，应该使用唯一的 ID 。

在第一类加速键定义中，加速键是双引号内区分大小写的字符：

```
"char", id      [.SHIFT] [.CONTROL] [.ALT]
```

如果你要使用该键与 Shift 、 Ctrl 和 ALT 中的一个或多个键的组合来定义加速键，那么只要将 SHIFT 、 CONTROL 或者 ALT 加入即可。

在第二类定义中，加速键是字符与 Ctrl 键的组合：

```
"^char", id      [.SHIFT] [.CONTROL] [.ALT]
```

当字符只与关键字 CONTROL 一起使用时，这种类型与第一种类型是相同的。

第三类和第四类使用一个数（ nCode ），而不是使用引号内的字符：

```
nCode, id, ASCII      [.SHIFT] [.CONTROL] [.ALT]
nCode, id, VIRTKEY    [.SHIFT] [.CONTROL] [.ALT]
```

这个数被解释为区分大小写的 ASCII 代码或者虚拟键码，这取决于关键字 ASCII 或者 VIRTKEY 。

最常用的加速键是第二类和第四类加速键。第二种类型是 Ctrl 键与字符键的组合。例如，下面的语句定义了 Ctrl-A 加速键：

```
"^A", id
```

第四种方法用于诸如功能键等虚拟键码，以下定义 Ctrl-F9 组合键为加速键：

```
VK_F9, wid, VIRTKEY, CONTROL
```

在 Windows 头文件中将标识符 VK_F9 定义为 F9 的虚拟键码，因此，在资源描述文件的开始处必须包含语句：

```
#include <windows.h>
```

上面所示的第一和第三种定义很少使用。如果你要使用它们，请注意区分字母的大小写。根据所按下的字符， Windows 对 " char " 或者 nCode 进行区分大小写的匹配。如果你加了 SHIFT 关键字，那么 Windows 检查 Shift 键是否被按下。这种情况有时会导致意料不到的结果。比如，如果 " char " 是 " A "，那么当按下 A 键的同时按下 Shift 键或者 Caps Lock 打开，但不同时使用 Shift 键和 Caps Lock 键时，则激活加速键。如果你与 SHIFT 一起使用 A 键，那么必须在 Shift 键按下时按 A ，但是在 Caps Lock 处于打开状态时，根本不会触发加速键。类似地， " a " 本身就是加速键，并且在 Caps Lock 处于打开状态时，按下 Shift 键的同时按下 A 键也得到加速键。不过，仅当 Shift 被按下并且 Caps Lock 也被打开时，带 Shift 的 " a " 才触发加速键。

当你为一个菜单项定义加速键时，应该将键的组合包含到菜单项的文本中。制表符（\t ）

将菜单项文本与加速键分隔开，将加速键排列在第二列。为了在菜单中给加速键作标记，可以使用文本 Ctrl、Shift 或者 Alt 之后跟上一个"+"号和一个键名。例如：

- F6
- Shift+F6
- Ctrl+F6

10.3.4 加速键表的装入

在你的程序中，你使用 LoadAccelerators 函数把加速键表装入内存，并获得该表的句柄。LoadAccelerators 语句非常类似于 LoadIcon、LoadCursor、LoadBitmap 和 LoadMenu 语句。

首先，把加速键表的句柄定义为类型 HANDLE：

```
HANDLE  hAccel;
```

然后装入加速键表：

```
hAccel = LoadAccelerators (hInstance, "MyAccelerator") ;
```

正如图标、光标、位图和菜单一样，你可以使用一个数值代替加速键表的名称，然后在 LoadAccelerators 语句中和 MAKEINTRESOURCE 宏一起使用该数值，或者把它放在双引号内，前面冠以字符#。

10.3.5 键盘代码转换

现在我们将讨论这样的三行代码，在本书中到目前为止创建的所有 Windows 程序中都使用过它们。这些代码是标准的消息循环：

```
while (GetMessage (&msg, NULL, 0, 0))
    {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
    }
```

下面修改这段代码，以便使用加速键：

```
while  (GetMessage (&msg, NULL, 0, 0))
    {
    if  (!TranslateAccelerator (hwnd, hAccel, &msg))
        {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
        }
    }
```

TranslateAccelerator 函数确定存放在 msg 消息结构中的消息是不是键盘消息。如果是，该函数将查找句柄为 hAccel 的加速键表。如果找到了一个匹配，则调用句柄为 hwnd 的窗口的窗口过程。如果加速键 ID 与系统菜单的菜单项一致，则消息就是 WM_SYSCOMMAND；否则，消息为 WM_COMMAND。

当 TranslateAccelerator 返回时，如果消息已经被转换（并且已经被发送给窗口过程），那么返回值为非零；否则，返回值为 0。如果 TranslateAccelerator 返回一个非零值，则不调用 TranslateMessage 和 DispatchMessage，而是经过循环回到 GetMessage 调用中。

TranslateAccelerator 中的参数 hwnd 看起来有点累赘，因为消息循环中的其它三个函数都没有要求这个参数。此外，消息结构本身（结构变量 msg）有一个叫做 hwnd 的域，它是窗口句柄。

msg 结构的域由 GetMessage 调用填充。当 GetMessage 的第二个参数为 NULL 时，函数检索应用程序所有窗口的消息。当 GetMessage 返回时，msg 结构的 hwnd 是将要获得消息的窗口的窗口句柄。然而，当 TranslateAccelerator 把键盘消息转换为 WM_COMMAND 或者 WM_SYSCOMMAND 消息时，它使用函数的第一个参数指定的窗口句柄 hwnd 来代替窗口句柄 msg.hwnd。Windows 就是这样把所有加速键消息发送给同一窗口过程的，即使另一个应用窗口当前拥有输入焦点。当模态对话框或者消息框拥有输入焦点时，TranslateAccelerator 不转换键盘消息，因为这些窗口的消息是不经过程序的消息循环的。

在某些情况下，当你的程序的另一个窗口（比如一个非模态对话框）拥有输入焦点时，你也许不想转换加速键。你将在下一章中看到如何处理这种情况。

10.3.6 接收加速键消息

当加速键与系统菜单中的菜单项相对应时，TranslateAccelerator 给窗口过程发送一条 WM_SYSCOMMAND 消息，否则，TranslateAccelerator 给窗口过程发送一条 WM_COMMAND 消息。下表所示为几种可能接收到的 WM_COMMAND 消息，这些消息用于加速键、菜单命令以及子窗口控制：

	LOWORD (wParam)	HIWORD (wParam)	lParam
加速键：	加速键 ID	1	0
菜单：	菜单 ID	0	0
控制：	控制 ID	通知码	子窗口句柄

如果加速键与一个菜单项对应，那么窗口过程还会收到 WM_INITMENU、WM_INITMENUPOPUP 和 WM_MENUSELECT 消息，就好像选中了菜单选项一样。在处理 WM_INITMENUPOPUP 时，程序往往启用和禁用弹出式菜单中的菜单项，因此，在使用加速键时，你仍然能够实现这类功能。如果加速键与一个禁用或者灰化的菜单项相对应，那么，TranslateAccelerator 函数就不会向窗口过程发送 WM_COMMAND 和 WM_SYSCOMMAND 消息。

如果活动窗口已经被最小化，那么 TranslateAccelerator 将为与启用的系统菜单项相对应的加速键向窗口过程发送 WM_SYSCOMMAND 消息，而不是 WM_COMMAND 消息。TranslateAccelerator 也会为没有任何菜单项与之对应的加速键而向窗口过程发送 WM_COMMAND 消息。

10.3.7 菜单与加速键应用程序 POPPAD

在第 8 章中，我们创建了一个叫做 POPPAD1 的程序，它使用了子窗口编辑控制来实现基本的笔记本功能。在这一章中，我们将加入 File 和 Edit 菜单，并称此程序为 POPPAD2 。Edit 菜单的菜单项的功能全部可用；我们将在第 11 章中完成 File 功能，在第 15 章中完成 Print 功能。 POPPAD2 如图 10-10 所示。

POPPAD2.MAK

```
#-----------------------
# POPPAD2.MAK make file
#-----------------------

poppad2.exe : poppad2.obj poppad2.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad2.exe poppad2.obj poppad2.res $(GUILIBS)

poppad2.obj : poppad2.c poppad2.h
     $(CC) $(CFLAGS) poppad2.c

poppad2.res : poppad2.rc poppad2.h poppad2.ico
     $(RC) $(RCVARS) poppad2.rc
```

POPPAD2.C

```
/*-----------------------------------------------------
   POPPAD2.C -- Popup Editor Version 2 (includes menu)
                (c) Charles Petzold, 1996
  -----------------------------------------------------*/

#include <windows.h>
#include "poppad2.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

char szAppName[] = "PopPad2" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HACCEL     hAccel ;
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, szAppName,
                          WS_OVERLAPPEDWINDOW,
                          GetSystemMetrics (SM_CXSCREEN) / 4,
                          GetSystemMetrics (SM_CYSCREEN) / 4,
                          GetSystemMetrics (SM_CXSCREEN) / 2,
                          GetSystemMetrics (SM_CYSCREEN) / 2,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hAccel = LoadAccelerators (hInstance, szAppName) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          if (!TranslateAccelerator (hwnd, hAccel, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          }
     return msg.wParam ;
     }

AskConfirmation (HWND hwnd)
     {
     return MessageBox (hwnd, "Really want to close PopPad2?",
                        szAppName, MB_YESNO | MB_ICONQUESTION) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndEdit ;
     int         iSelect, iEnable ;

     switch (iMsg)
          {
          case WM_CREATE :
               hwndEdit = CreateWindow ("edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                              WS_BORDER | ES_LEFT | ES_MULTILINE |
                              ES_AUTOHSCROLL | ES_AUTOVSCROLL,
```

```
                0, 0, 0, 0,
                hwnd, (HMENU) 1,
                ((LPCREATESTRUCT) lParam)->hInstance, NULL) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hwndEdit) ;
          return 0 ;

     case WM_SIZE :
          MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                                      HIWORD (lParam), TRUE) ;
          return 0 ;

     case WM_INITMENUPOPUP :
          if (lParam == 1)
               {
               EnableMenuItem ((HMENU) wParam, IDM_UNDO,
                    SendMessage (hwndEdit, EM_CANUNDO, 0, 0) ?
                         MF_ENABLED : MF_GRAYED) ;

               EnableMenuItem ((HMENU) wParam, IDM_PASTE,
                    IsClipboardFormatAvailable (CF_TEXT) ?
                         MF_ENABLED : MF_GRAYED) ;

               iSelect = SendMessage (hwndEdit, EM_GETSEL, 0, 0) ;

               if (HIWORD (iSelect) == LOWORD (iSelect))
                    iEnable = MF_GRAYED ;
               else
                    iEnable = MF_ENABLED ;

               EnableMenuItem ((HMENU) wParam, IDM_CUT,   iEnable) ;
               EnableMenuItem ((HMENU) wParam, IDM_COPY,  iEnable) ;
               EnableMenuItem ((HMENU) wParam, IDM_DEL,   iEnable) ;

               return 0 ;
               }
          break ;

     case WM_COMMAND :

          if (lParam)
               {
               if (LOWORD (lParam) == 1 &&
                    (HIWORD (wParam) == EN_ERRSPACE ||
                     HIWORD (wParam) == EN_MAXTEXT))
                         MessageBox (hwnd,
                                   "Edit control out of space.",
                                   szAppName, MB_OK | MB_ICONSTOP) ;
               return 0 ;
               }

          else switch (LOWORD (wParam))
                    {
```

```
                    case IDM_NEW :
                    case IDM_OPEN :
                    case IDM_SAVE :
                    case IDM_SAVEAS :
                    case IDM_PRINT :
                         MessageBeep (0) ;
                         return 0 ;

                    case IDM_EXIT :
                         SendMessage (hwnd, WM_CLOSE, 0, 0) ;
                         return 0 ;

                    case IDM_UNDO :
                         SendMessage (hwndEdit, WM_UNDO, 0, 0) ;
                         return 0 ;

                    case IDM_CUT :
                         SendMessage (hwndEdit, WM_CUT, 0, 0) ;
                         return 0 ;

                    case IDM_COPY :
                         SendMessage (hwndEdit, WM_COPY, 0, 0) ;
                         return 0 ;

                    case IDM_PASTE :
                         SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
                         return 0 ;

                    case IDM_DEL :
                         SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
                         return 0 ;

                    case IDM_SELALL :
                         SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
                         return 0 ;

                    case IDM_HELP :
                         MessageBox (hwnd, "Help not yet implemented!",
                              szAppName, MB_OK | MB_ICONEXCLAMATION) ;
                         return 0 ;

                    case IDM_ABOUT :
                         MessageBox (hwnd,
                              "POPPAD2 (c) Charles Petzold, 1996",
                              szAppName, MB_OK | MB_ICONINFORMATION) ;
                         return 0 ;
                    }
               break ;

          case WM_CLOSE :
               if (IDYES == AskConfirmation (hwnd))
                    DestroyWindow (hwnd) ;
               return 0 ;

          case WM_QUERYENDSESSION :
```

```
            if (IDYES == AskConfirmation (hwnd))
                 return 1 ;
            else
                 return 0 ;

       case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
       }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
  }
```

POPPAD2.RC

```
/*----------------------------
   POPPAD2.RC resource script
  ----------------------------*/

#include <windows.h>
#include "poppad2.h"

PopPad2 ICON poppad2.ico

PopPad2 MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New",                IDM_NEW
          MENUITEM "&Open... ",           IDM_OPEN
          MENUITEM "&Save",               IDM_SAVE
          MENUITEM "Save &As... ",        IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print",              IDM_PRINT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",               IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",       IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t\tCtrl+X",        IDM_CUT
          MENUITEM "&Copy\tCtrl+C",       IDM_COPY
          MENUITEM "&Paste\tCtrl+V",      IDM_PASTE
          MENUITEM "De&lete\tDel",        IDM_DEL
          MENUITEM SEPARATOR
          MENUITEM "&Select All",         IDM_SELALL
          }
     POPUP "&Help"
          {
          MENUITEM "&Help... ",           IDM_HELP
          MENUITEM "&About PopPad2... ", IDM_ABOUT
          }
     }

PopPad2 ACCELERATORS
```

```
{
"^Z",        IDM_UNDO
VK_BACK,     IDM_UNDO,   VIRTKEY, ALT
"^X",        IDM_CUT
VK_DELETE,   IDM_CUT,    VIRTKEY, SHIFT
"^C",        IDM_COPY
VK_INSERT,   IDM_COPY,   VIRTKEY, CONTROL
"^V",        IDM_PASTE
VK_INSERT,   IDM_PASTE,  VIRTKEY, SHIFT
VK_DELETE,   IDM_DEL,    VIRTKEY
VK_F1,       IDM_HELP,   VIRTKEY
}
```

POPPAD2.H

```
/*-----------------------
   POPPAD2.H header file
  -----------------------*/

#define IDM_NEW      1
#define IDM_OPEN     2
#define IDM_SAVE     3
#define IDM_SAVEAS   4
#define IDM_PRINT    5
#define IDM_EXIT     6

#define IDM_UNDO     10
#define IDM_CUT      11
#define IDM_COPY     12
#define IDM_PASTE    13
#define IDM_DEL      14
#define IDM_SELALL   15

#define IDM_HELP     20
#define IDM_ABOUT    22
```

POPPAD2.ICO

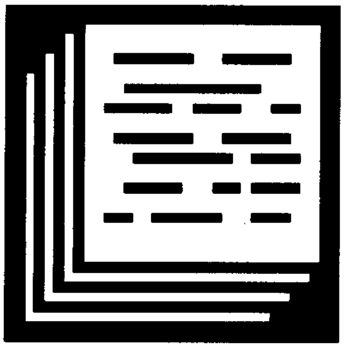


图 10-10 POPPAD2程序

POPPAD2.RC 资源描述文件包含菜单和加速键。你将注意到，所有加速键都表示在制表符（\t）后的 Edit 弹出式菜单字符串中。

10.3.8 启用菜单项

窗口过程的工作包括启用和灰化 Edit 菜单中的选项，这项工作在处理 WM_INITMENUPOPUP 时完成。首先，程序检查是否要显示 Edit 弹出式菜单。因为菜单里 Edit 的位置索引（File 从 0 开始）是 1，因此如果即将显示 Edit 弹出式菜单，那么 lParam 应该等于 1。

为了确定是否启用 Undo，POPPAD2 给编辑控制发送一条 EM_CANUNDO 消息。如果编辑控制能够执行 Undo 动作，那么 SendMessage 调用返回非零值。在这种情况下，选项被启用；否则，选项灰化。

```
EnableMenuItem ( (HMENU)wParam, IDM_UNDO,
    SendMessage (hwndEdit, EM_CANUNDO, 0, 0) ?
          MF_ENABLED : MF_GRAYED) ;
```

只有当剪贴板中包含文本时，Paste 选项才能够被启用。我们可以使用 CF_TEXT 标识符通过 IsClipboardFormatAvailable 调用来确定这一点：

```
EnableMenuItem ( (HMENU)wParam, IDM_PASTE,
    IsClipboardFormatAvailable (CF_TEXT) ?
        MF_ENABLED : MF_GRAYED) ;
```

只有选择了编辑控制中的文本，Cut、Copy 和 Delete 选项才能够被启用。给编辑控制发送一条 EM_GETSEL 消息，并返回包含此信息的整数：

```
iSelect = SendMessage (hwndEdit, EM_GETSEL, 0, 0) ;
```

iSelect 的低位字是第一个被选中字符的位置，iSelect 的高位字是下一个被选中字符的位置。如果这两个字相等，则表明没有选中文本：

```
if (HIWORD (iSelect) == LOWORD (iSelect))
    iEnable = MF_GRAYED;
else
    iEnable = MF_ENABLED;
```

然后可以将 iEnable 的值用于 Cut、Copy 和 Delete 选项：

```
EnableMenuItem ((HMENU)wParam, IDM_CUT,  iEnable) ;
EnableMenuItem ((HMENU)wParam, IDM_COPY, iEnable) ;
EnableMenuItem ((HMENU)wParam, IDM_DEL,  iEnable) ;
```

10.3.9 处理菜单项

当然，如果对 POPPAD2 程序不使用子窗口编辑控制，那么我们将面临一些问题，这

涉及如何完成 Edit 菜单中的 Undo 、 Cut 、 Copy 、 Paste 、 Clear 和 SelectAll 选项。正是编辑控制使得这种处理变得容易，因为我们对每一个选项只需向编辑控制发送一条消息即可：

```
case IDM_UNDO :
     SendMessage (hwndEdit, WM_UNDO, 0, 0)
     return 0 ;

case IDM_CUT :
     SendMessage (hwndEdit, WM_CUT, 0, 0) ;
     return 0 ;

case IDM_COPY :
     SendMessage (hwndEdit, WM_COPY, 0, 0) ;
     return 0 ;

case IDM_PASTE :
     SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
     return 0 ;

case IDM_DEL :
     SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
     return 0 ;

case IDM_SELALL :
     SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
     return 0 ;
```

注意，我们可以更进一步简化这些处理——只要使 IDM_UNDO 、 IDM_CUT 等的值等于相应的窗口消息 WM_UNDO 、 WM_CUT 等的值。

File 弹出式菜单上的 About 选项激活一个简单的消息框：

```
case IDM_ABOUT:
    MessageBox (hwnd,
        "POPPAD2  (c) Charles Petzold, 1996",
        szAppName, MB_OK | MB_ICONINFORMATION) ;
    return 0;
```

在第 11 章中，我们将把它变成一个对话框。当你从菜单中选择 Help 选项或者按下 F1 加速键时，同样可以激活一个消息框。

Exit 选项向窗口过程发送一条 WM_CLOSE 消息：

```
case IDM_EXIT:
    SendMessage (hwnd, WM_CLOSE, 0, 0) ;
    return 0;
```

这正是 DefWindowProc 收到一条 wParam 等于 SC_CLOSE 的 WM_SYSCOMMAND 消息时所完成的工作。

在前面的那些程序中，我们没有在窗口过程中处理 WM_CLOSE 消息，而只是简单地把它送给 DefWindowProc 。 DefWindowProc 对 WM_CLOSE 的处理非常简单：调用 DestroyWindows 函数。可以不把 WM_CLOSE 消息送给 DefWindowProc ，而让 POPPAD2 来处理它。这个事实现在并不重要，但是在第 11 章中当 POPPAD 可以真正编辑文本时，它就变得非常重要了。

```
case WM_CLOSE:
    if (IDYES = AskConfirmation (hwnd))
        DestroyWindow (hwnd) ;
    return 0;
```

AskConfirmation 是 POPPAD2 中的一个函数，它显示一个请求确认关闭程序的消息框：

```
AskConfirmation (HWND hwnd)
    {
    return MessageBox (hwnd, "Do you really want to close Poppad2?",
                       szAppName, MB_YESNO | MB_ICONQUESTION) ;
    }
```

消息框（以及 AskConfirmation 函数）返回 IDYES ，如果选择了 YES 按钮的话。只有这样程序才调用 DestroyWindow ，否则，程序并不结束。

如果在程序结束之前你想确认某种操作，那么你还必须处理 WM_QUERYENDSESSION 消息。当用户结束一个 Windows 执行时， Windows 开始向每个窗口过程发送一个 WM_QUERYENDSESSION 消息。如果有任何一个窗口过程从这个消息中返回 0 ，那么 Windows 将不会结束。我们如下处理 WM_QUERYENDSESSION：

```
case WM_QUERYENDSESSION:
    if (IDYES == AskConfirmation (hwnd))
        return 1;
    else
        return 0;
```

如果在程序结束之前要求用户的确认，只需处理 WM_CLOSE 和 WM_QUERYENDSESSION 这两条消息，这就是为什么我们使 POPPAD2 中的 Exit 菜单选项只向窗口过程发送一条 WM_CLOSE 消息的原因。这样做，我们避免了在第三处进行请求确认的操作。

如果要处理 WM_QUERYENDSESSION 消息，那么你也许还会对 WM_ENDSESSION 消息感兴趣。 Windows 把这条消息发送给先前收到 WM_QUERYENDSESSION 消息的每个窗口过程。如果由于另一个程序从 WM_QUERYENDSESSION 返回了 0 而不能结束 Windows 的运行，那么 WM_ENDSESSION 的 wParam 参数为 0 。 WM_ENDSESSION 消息实际上回答了这个问题：我告诉过 Windows 可以结束，但是我真的结束了吗？

尽管在 POPPAD2 的 File 菜单中我包括了常见的 New 、 Open 、 Save 和 Save As 选项，但是它们现在并不起作用。要处理这些命令，我们需要使用对话框。现在是讨论对话框的时候了。

第 11 章　对 话 框

如果有很多输入从而超出了菜单可以处理的程度，那么我们使用对话框来获取输入。程序员可以通过在某项后面加上省略号（…）来表示该菜单项将激活一个对话框。

对话框的一般形式是包含多种子窗口控制的弹出式窗口，这些控制的大小和位置在程序资源描述文件的“对话框模板”中指定。Microsoft Windows 95 负责创建弹出式对话框窗口和子窗口控制，并提供处理对话框消息（包括所有键盘和鼠标输入）的窗口过程。有时候称完成这些功能的 Windows 内部代码为“对话框管理器”。

Windows 的内部对话框窗口过程所处理的许多消息也传递给你自己程序中的函数，这个函数即是所谓的“对话框过程”或者“对话过程”。该函数与通常的窗口过程类似，但是也存在一些重要区别。一般说来，除了在创建对话框时初始化子窗口控制，处理来自子窗口控制的消息以及结束对话框之外，程序员不需要再给对话框过程增加其它功能。

对话框这个主题的含义太广了，因为它还包含子窗口控制的使用。不过，我们已经在第 8 章中研究了子窗口控制。当你在对话框中使用子窗口控制时，第 8 章所提到的许多工作可以由 Windows 的对话框管理器来完成。尤其是，在程序 COLORS1 中遇到的在滚动条之间切换输入焦点的问题也不会在对话框中出现。Windows 会处理对话框中的控制之间切换输入焦点所必需完成的所有工作。

不过，在程序中加入对话框并不是一件轻而易举的事，它还涉及到对几个文件的修改——对话框模板放在资源描述文件中，对话框过程放在源代码中，对话框中所使用的标识符一般放在程序的头文件中。我们将从一个简单的对话框开始，让读者对各部分之间的相互联系有所了解。

11.1　模态对话框

对话框分两类：“模态的”和“非模态的”。模态对话框最为普遍。当你的程序显示一个模态对话框时，用户不能在对话框与同一个程序中的另一个窗口之间进行切换，用户必须显式地结束该对话框，这通常通过按一下标记为 OK 或者 Cancel 的压入按钮来完成。不过，在显示模态对话框时，用户通常可以从当前程序切换到另一个程序。有些对话框（称为“系统模态”）甚至连这样的操作也不允许。在 Windows 中，显示了系统模态对话框之后，要完成其它任何工作，必须先结束该对话框。

11.1.1 创建一个“About”对话框

Windows 程序即使不需要接收用户输入，也通常具有一个由菜单上的 About 选项激活的对话框，该对话框用来显示程序的名字、图标、版权标志和标记为 OK 的压入按钮，也许还会有其它信息。我们将要看到的第一个程序除了显示一个 About 对话框外，别无它用。这个 ABOUT1 程序如图 11-1 所示：

ABOUT1.MAK

```
#----------------------
# ABOUT1.MAK make file
#----------------------

about1.exe : about1.obj about1.res
     $(LINKER) $(GUIFLAGS) -OUT:about1.exe about1.obj about1.res $(GUILIBS)

about1.obj : about1.c about1.h
     $(CC) $(CFLAGS) about1.c

about1.res : about1.rc about1.h about1.ico
     $(RC) $(RCVARS) about1.rc
```

ABOUT1.C

```
/*------------------------------------------
   ABOUT1.C -- About Box Demo Program No. 1
               (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about1.h"

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About1" ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
```

```
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "About Box Demo Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static WNDPROC   lpfnAboutDlgProc ;
     static HINSTANCE hInstance ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_ABOUT :
                         DialogBox (hInstance, "AboutBox", hwnd, AboutDlgProc) ;
                         return 0 ;
                    }
               break ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam,
                                                  LPARAM lParam)
     {
     switch (iMsg)
```

```
          {
          case WM_INITDIALOG :
               return TRUE ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDOK :
                    case IDCANCEL :
                         EndDialog (hDlg, 0) ;
                         return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }
```

ABOUT1.RC

```
/*---------------------------
   ABOUT1.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about1.h"

About1 ICON about1.ico

About1 MENU
     {
     POPUP "&Help"
          {
          MENUITEM "&About About1...",       IDM_ABOUT
          }
     }

AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "About1"                         -1,    0, 12, 160,  8
     ICON  "About1"                         -1,    8,  8,   0,  0
     CTEXT "About Box Demo Program"         -1,    0, 36, 160,  8
     CTEXT "(c) Charles Petzold, 1996"      -1,    0, 48, 160,  8
     DEFPUSHBUTTON "OK"                   IDOK,   64, 60,  32, 14, WS_GROUP
     }
```

ABOUT1.H

```
/*-----------------------
   ABOUT1.H header file
  -----------------------*/

#define IDM_ABOUT       1
```

ABOUT1.ICO

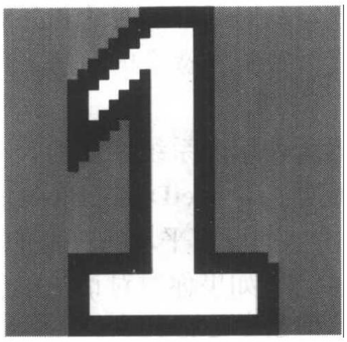


图 11-1　ABOUT1程序

11.1.2　对话框模板

向程序中加入对话框时，要做的第一件事是设计对话框模板。这个模板可以直接放在资源描述文件中，也可以放在一个单独的文件中。按照约定，该文件使用扩展名.DLG（代表“ dialog ”）。如果程序员将模板放在文件 filename.dlg 里，则在资源描述文件中应该有下面这条语句：

```
rcinclude filename.dlg
```

你可以在文本编辑器里手工创建对话框模板，或者使用任何一个自动工具。这些工具的输出质量并不好，因此我给出的对话框模板看起来像是用手工创建的。

ABOUT1 的对话框模板如下所示：

```
AboutBox DIALOG 20, 20, 160, 80
    STYLE WS_POPUP | WS_DLGFRAME
    {
    CTEXT "About1"                          -1,   0, 12, 160,  8
    ICON  "About1"                          -1,   8,  8,   0,  0
    CTEXT "About1 Box Demo Program"         -1,   0, 36, 160,  8
    CTEXT "(c) Charles Petzold, 1996"       -1,   0, 48, 160,  8
    DEFPUSHBUTTON "OK"                    IDOK,  64, 60,  32, 14, WS_GROUP
    }
```

第一行给对话框一个名字（在这种情况下，为 AboutBox ）。对于其它资源，程序员可以使用一个数字来代替其名字。名字后面跟着关键字 DIALOG 和四个数字。前两个数是对话框左上角的 x 和 y 坐标，即激活对话框时它在其父窗口客户区中的位置；后面两个数是对话框的宽度和高度。

这些坐标和大小的单位不是象素点，它们是基于仅用于对话框模板的专用坐标系，数的单位是根据系统字体字符的大小来选取，x 坐标和宽度的单位为平均字符宽度的 1/4 ，y 坐标和高度的单位为平均字符高度的 1/8 。因此，对于这里的对话框，对话框的左上角距离主窗口客户区的左边有 5 个字符宽，距离上边有 2.5 个字符宽，对话框的宽度为 40 个字

符，高为 10 个字符。

这样的坐标系使得程序员可以使用坐标和大小来大致勾勒对话框的尺寸和外观，而不管视频显示器的分辨率是多少。由于系统字体字符的高度大致为其宽度的两倍，所以， x 轴和 y 轴的量度差不多相等。

函数 GetDialogBaseUnits 使你可以确定对话框管理器所使用的系统字体的大小。对于标准的 VGA （这也许是 Windows 使用最广泛的显示适配器）， GetDialogBaseUnits 返回的字符宽度为 8 ，字符高度为 16 。由于在对话框的单位中，宽度为平均字符宽的 1/4 ，高度为 1/8 ，所以每个对话框单位对应于 VGA 上的两个象素点。如果你对对话框单位的算法感到太抽象，那么不妨使用上面的这种方法来进行计算，但是不要依赖它。

模板中的 STYLE 语句类似于 CreateWindow 调用中的风格域。对于模态对话框，通常使用 WS_POPUP 和 WS_DLGFRAME ，我们将在以后介绍一些其它的选项。

在左右花括号之间，定义出现在对话框中的子窗口控制。这个对话框使用了三种类型的子窗口控制，它们分别是 CTEXT （文本居中）、 ICON （图标）和 DEFPUSHBUTTON （缺省压入按钮）。这些语句的格式为：

```
control-type "text" id, xPos, yPos, xWidth, yHeight [, iStyle]
```

其中， iStyle 项是可选的，它使用 Windows 头文件中定义的标识符来指定其它窗口风格。

CTEXT 、 ICON 和 DEFPUSHBUTTON 等标识符只可以在对话框中使用，它们是某种特定窗口风格的缩写。例如， CTEXT 指示这个子窗口控制类是“静态的”，其风格为：

```
WS_CHILD | SS_CENTER | WS_VISIBLE | WS_GROUP
```

虽然前面没有出现过 WS_GROUP 标识符，但是我们在第 8 章的 COLORS1 程序中已经出现过 WS_CHILD 、 SS_CENTER 和 WS_VISIBLE 窗口风格，在创建静态子窗口文本控制时已经用到了它们。

对于图标，文本字段是程序的图标资源名，它也在 ABOUT1 资源描述文件中定义。对于压入按钮，文本字段是出现在压入按钮里的文本，这个文本等价于在程序中创建子窗口控制时调用 CreateWindow 所指定的第二个参数。

id 字段是子窗口在向其父窗口发送消息（通常为 WM_COMMAND 消息）时用来标识它自身的值。这些子窗口控制的父窗口就是对话框本身，它将这些消息发送给 Windows 的一个窗口过程。不过，这个窗口过程也将这些消息发送给你在程序中给出的对话框过程。id 值等价于我们在第 8 章创建子窗口时，在 CreateWindow 函数中使用的子窗口 ID 。由于文本和图标控制不向父窗口回送消息，所以这些值被设置为-1 。压入按钮的 id 值为 IDOK ，它在 Windows 头文件中被定义为 1 。

接下去的四个数设置子窗口的位置（相对于对话框客户区的左上角）和大小，它们是以系统字体平均宽度的 1/4 和平均高度的 1/8 为单位来表示的。对于 ICON 语句，宽度和高度将被忽略。

对话框模板中的 DEFPUSHBUTTON 语句，除了包含 DEFPUSHBUTTON 关键字所隐含的窗口风格，还包含窗口风格 WS_GROUP 。在后面讨论该程序的第二个版本 ABOUT2

时，还要详细说明 WS_GROUP（以及相关的 WS_TABSTOP 风格）。

11.1.3 对话框过程

你程序内的对话框过程处理传给对话框的消息。尽管看起来很像是窗口过程，但是它并不是真实的窗口过程。对话框的窗口过程在 Windows 内部定义，这个窗口过程调用你编写的对话框过程，把它所接收到的许多消息作为参数。下面是 ABOUT1 的对话框过程：

```
BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_INITDIALOG :
               return TRUE ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDOK :
                    case IDCANCEL :
                         EndDialog (hDlg, 0) ;
                         return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }
```

该函数的参数与常规窗口过程的参数相同；与窗口过程类似，对话框过程必须定义为一个 CALLBACK（回调）函数。尽管我使用了 hDlg 作为对话框窗口的句柄，但是你也可以按照你自己的意愿使用 hwnd。首先，让我们来看一下这个函数与窗口过程的区别：

- 窗口过程返回一个 LRESULT；对话框返回一个 BOOL（在 Windows 头文件中，BOOL 被定义为 int 类型）。
- 如果窗口过程不处理某个特定的消息，那么它将调用 DefWindowProc；如果对话框过程处理一条消息，那么它返回 TRUE（非零），如果不处理则返回 FALSE（0）。
- 对话框过程不需要处理 WM_PAINT 或 WM_DESTROY 消息。对话框过程不接收 WM_CREATE 消息；而是在特殊的 WM_INITDIALOG 消息期间，对话框过程执行初始化操作。

WM_INITDIALOG 消息是对话框接收到的第一条消息，这条消息只发送给对话框过程。如果对话框过程返回 TRUE，那么 Windows 将输入焦点设置为对话框中第一个具有 WS_TABSTOP 风格（我们将在 ABOUT2 的讨论中加以解释）的子窗口控制。在这个对话框中，第一个具有 WS_TABSTOP 风格的子窗口控制是压入按钮。另外，对话框过程也可以在处理 WM_INITDIALOG 时使用 SetFocus 来将输入焦点设置为对话框中的某个子窗口控制，然后返回 FALSE。

此外，对话框过程只处理 WM_COMMAND 消息。这是当压入按钮被鼠标点中，或者

如果在按钮具有输入焦点的情况下按下 Spacebar 时，压入按钮控制发送给其父窗口的消息。这个控制的 ID（我们在对话框模板中将其设置为 IDOK）在 wParam 的低位字中，对于这条消息，对话框过程调用 EndDialog，它告诉 Windows 清除对话框。对于所有其它消息，对话框过程返回 FALSE，并告诉 Windows 内部的对话框窗口过程：我们的对话框过程不处理这些消息。

模态对话框的消息不通过你程序的消息队列，所以不必担心对话框中键盘加速键的影响。

11.1.4 激活对话框

在 WndProc 中处理 WM_CREATE 消息时，ABOUT1 获取程序的实例句柄并将它放在静态变量中：

```
hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
```

ABOUT1 检查 WM_COMMAND 消息，以确保消息 wParam 的低位字等于 IDM_ABOUT。当它获得这样的一个消息时，程序调用 DialogBox：

```
DialogBox (hInstance, "AboutBox", hwnd, AboutDlgProc) ;
```

该函数需要实例句柄（在处理 WM_CREATE 时保存的）、对话框名（在资源描述文件中定义的）、对话框的父窗口（也是程序的主窗口）和对话框过程的地址。如果你使用一个数字而不是对话框模板名，那么可以用 MAKEINTRESOURCE 宏将它转换为一个串。

从菜单中选择"About About1..."将显示如图 11-2 中所示的对话框。你可以用鼠标按一下 OK 按钮、按 Spacebar 或者按 Enter 来结束这个对话框。对于任意包含缺省压入按钮的对话框，在按 Enter 或者 Spacebar 之后，Windows 给对话框发送一条 WM_COMMAND 消息，并令 wParam 的低位字等于缺省压入按钮的 ID。

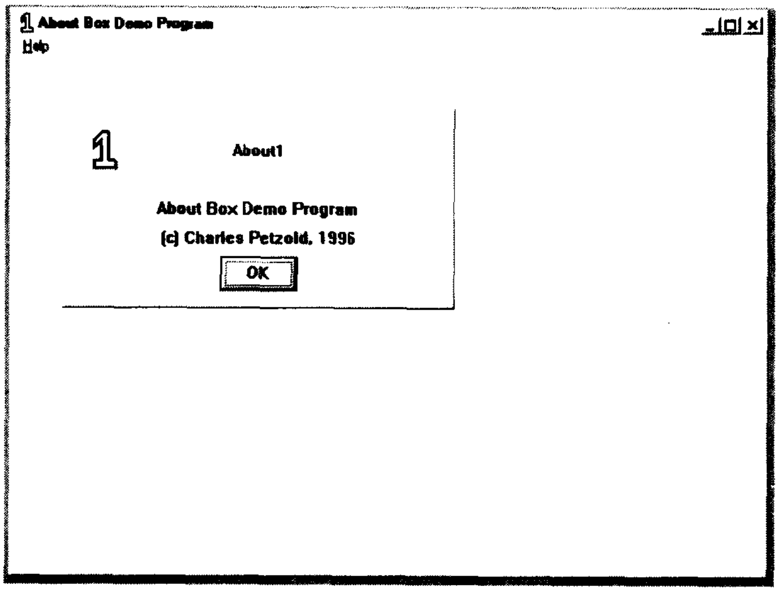


图 11-2 ABOUT1 程序的对话框

直到对话框结束之后，你用来显示对话框的 DialogBox 才将控制返回给 WndProc 。DialogBox 的返回值是对话框过程内部调用的 EndDialog 函数的第二个参数（这个值未在 ABOUT1 中使用，但是在 ABOUT2 中使用了）。然后， WndProc 可以将控制返回给 Windows 。

即使在显示对话框时， WndProc 也可以继续接收消息。实际上，你可以从对话框过程内部给 WndProc 发送消息。 ABOUT1 的主窗口是弹出式对话框窗口的父窗口，所以 AboutDlgProc 中的 SendMessage 调用可以使用如下语句开始：

```
SendMessage (GetParent (hDlg), ...) ;
```

11.1.5 关于对话框风格的补充说明

对话框的窗口风格在该对话框模板的 STYLE 语句中指定。对于 ABOUT1 ，我们使用一种对于模态对话框而言最为普遍的风格：

```
STYLE  WS_POPUP | WS_DLGFRAME
```

然而，你也可以尝试其它风格，例如，你可以尝试：

```
STYLE WS_POPUP | WS_CAPTION
```

这样将创建一个带有标题条（ caption bar ）和普通窗口边框的对话框。标题条允许用户使用鼠标在显示屏上移动对话框。如果你使用 WS_CAPTION ，那么 DIALOG 语句中所指定的 x 坐标和 y 坐标是对话框客户区的坐标，并相对于父窗口客户区的左上角。标题条将在 y 坐标之上显示。

如果你使用了标题条，那么可以在对话框模板中使用 CAPTION 语句，以便在标题条中放入文本：

```
CAPTION "Dialog Box Caption"
```

该语句跟在 STYLE 语句之后；或者，在对话框过程中处理 WM_INITDIALOG 消息时，你可以使用：

```
SetWindowText (hDlg, "Dialog Box Caption") ;
```

如果你使用 WS_CAPTION 风格，那么还可以用 WS_SYSMENU 风格来增加一个系统菜单框：

```
STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU
```

该风格允许用户从系统菜单中选择 Move 或者 Close 。

在风格中加入 WS_THICKFRAME 时，用户就可以改变对话框的大小。不过，对于对话框来说，改变大小是很罕见的。如果你希望该对话框不同寻常，那么还可以尝试在 STYLE 语句中加入 WS_MAXIMIZEBOX 。

STYLE 语句并不是必需的。如果你在模板中不包含 STYLE 或者 CAPTION 语句，那么缺省风格为：

```
WS_POPUP | WS_BORDER
```

但是，这样看起来总有些呆板。WS_DLGFRAME 产生的效果会更具有吸引力。如果你在 STYLE 语句中包含了 CAPTION 语句，那么缺省风格为：

```
WS_POPUP | WS_CAPTION | WS_SYSMENU
```

你还可以在对话框模板中指定：

```
MENU menu-name
```

从而在对话框中加入菜单，其中的参数可以是资源描述文件中的菜单名字或者菜单号。模态对话框很少使用菜单。如果使用了菜单，那么你必须确保菜单和对话框中的所有 ID 号都是唯一的。

FONT 语句使你可以设置非系统字体，以供对话框文本使用。

尽管对话框窗口过程通常位于 Windows 内部，但是你也可以使用自己编写的窗口过程来处理对话框消息。要这样做，你必须在对话框模板中指定一个窗口类名：

```
CLASS "class-name"
```

这种用法很少见，但是在本章后面所示的 HEXCALC 程序中我们将用到它。

当你使用对话框模板的名字来调用 DialogBox 时，Windows 通过调用普通的 CreateWindow 函数来完成创建弹出式窗口所需要完成的一切操作。Windows 从对话框模板中获取窗口的坐标、大小、窗口风格、标题和菜单，从 DialogBox 的参数中获得实例句柄和父窗口句柄。它所需要的唯一其它信息是一个窗口类（假设对话框模板不指定窗口类的话）。Windows 为对话框注册一个专用的窗口类，这个窗口类的窗口过程可以访问对话框过程指针（该指针是你在 DialogBox 调用中指定的），所以它可以使程序获得该弹出式窗口所接收的消息。当然，你可以通过自己创建弹出式窗口来创建和维护自己的对话框。不过，使用 DialogBox 则更简单。

11.1.6 关于定义控制的补充说明

在 ABOUT1.RC 的对话框模板中，我们使用缩写 CTEXT、ICON 和 DEFPUSHBUTTON 来定义对话框所需要的三种类型的子窗口控制。你还可以使用其它类型，每种类型隐含一个特定的预定义窗口类和一种窗口风格。下表显示了每种控制类型的等价窗口类和窗口风格：

控制类型	窗口类	窗口风格
PUSHBUTTON	按钮	BS_PUSHBUTTON | WS_TABSTOP
DEFPUSHBUTTON	按钮	BS_DEFPUSHBUTTON | WS_TABSTOP
CHECKBOX	按钮	BS_CHECKBOX | WS_TABSTOP
RADIOBUTTON	按钮	BS_RADIOBUTTON | WS_TABSTOP
GROUPBOX	按钮	BS_GROUPBOX | WS_TABSTOP
LTEXT	静态	SS_LEFT | WS_GROUP
CTEXT	静态	SS_CENTER | WS_GROUP

续表

控制类型	窗口类	窗口风格
RTEXT	静态	SS_RIGTH I WS_GROUP
ICON	静态	SS_ICON
EDITTEXT	编辑	SS_LEFT I WS_BORDER I WS_TABSTOP
SCROLLBAR	滚动条	SBS_HORZ
LISTBOX	列表框	LBS_NOTIFY I WS_BORDER I WS_VSCROLL
COMBOBOX	组合框	CBS_SIMPLE I WS_TABSTOP

RC 资源编译器是唯一能够识别这些缩写的程序。除了表中所示的窗口风格外，每个控制还具有风格：

```
WS_CHILD | WS_VISIBLE
```

对于这些控制类型，除了 EDITTEXT 、 SCROLLBAR 、 LISTBAR 和 COMBOBOX 外，控制语句的格式为：

```
control-type "text", id, xPos, yPos, xWidth, yHeight, [, iStyle]
```

对于 EDITTEXT 、 SCROLLBAR 、 LISTBAR 和 COMBOBOX ，其格式为：

```
control-type id, xPos, yPos, xWidth, yHeight, [, iStyle]
```

其中没有文本字段。在这两种语句中， iStyle 参数都是可选的。

在第 8 章中，我讨论了确定预定义子窗口的宽度和高度的规则。你可能需要回到第 8 章去参考这些规则，这时请记住：对话框中指定大小的单位为平均字符宽度的 1/4 ，平均字符高度的 1/8 。

控制语句的"style"字段是可选的。它允许你包含其它窗口风格标识符，例如，如果你想创建在正方形框左边包含文本的复选框，那么可以使用：

```
CHECKBOX "text", id, xPos, yPos, xWidth, yHeight, BS_LEFTTEXT
```

虽然使用子窗口控制的缩写表示很方便，但它并不完整，例如，你无法创建没有边框的子窗口编辑控制。为此， RC 资源编译器还识别一个通用的控制语句，其格式如下：

```
CONTROL "text", id, "class", iStyle, xPos, yPos, xWidth, yHeight
```

这个语句允许通过指定窗口类和完整的窗口风格来创建任意类型的子窗口控制。例如，你可以使用：

```
CONTROL "OK", IDOK, "button", WS_CHILD | WS_VISIBLE |
           BS_PUSHBUTTON | WS_TABSTOP, 10, 20, 32, 14
```

来代替

```
PUSHBUTTON "OK", IDOK, 10, 20, 32, 14
```

当编译完资源描述文件时，这两条语句在.RES 和.EXE 文件中的编码是相同的。

当你在对话框模板中使用 CONTROL 语句时，不必包含 WS_CHILD 和 WS_VISIBLE

风格。 Windows 在说明控制格式的同时还说明了 Windows 对话框管理器在创建对话框时所完成的操作。首先，它创建一个弹出式窗口，其父窗口句柄在 DialogBox 函数中提供，这一点我在前面已经讲过。然后，对话框管理器为对话框模板中的每个控制创建一个子窗口。所有这些控制的父窗口均是这个弹出式对话框。上面给出的 CONTROL 语句被转换成一个 CreateWindow 调用，形式如下所示：

```
CreateWindow ("button", "OK",
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
    10 * cxChar / 4, 20 * cyChar / 8,
    32 * cxChar / 4, 14 * cyChar / 8,
    hDlg, IDOK, hInstance NULL) ;
```

其中， cxChar 和 cyChar 是系统字体字符的宽度和高度，以象素为单位； hDlg 是从创建该对话框窗口的 CreateWindow 调用返回的值； hInstance 参数是从 DialogBox 调用获得的。

11.1.7 一个更复杂的对话框

ABOUT1 中的简单对话框演示了设计和运行一个对话框的要点；现在让我们来看一个稍微复杂些的例子。图 11-3 给出的 ABOUT2 程序演示了如何在对话框过程中管理控制以及如何在对话框的客户区中绘图。

ABOUT2.MAK

```
#----------------------
# ABOUT2.MAK make file
#----------------------

about2.exe : about2.obj about2.res
     $(LINKER) $(GUIFLAGS) -OUT:about2.exe about2.obj about2.res $(GUILIBS)

about2.obj : about2.c about2.h
     $(CC) $(CFLAGS) about2.c

about2.res : about2.rc about2.h about2.ico
     $(RC) $(RCVARS) about2.rc
```

ABOUT2.C

```
/*------------------------------------------
   ABOUT2.C -- About Box Demo Program No. 2
               (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about2.h"

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
int iCurrentColor  = IDD_BLACK,
    iCurrentFigure = IDD_RECT ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About2" ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "About Box Demo Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void PaintWindow (HWND hwnd, int iColor, int iFigure)
     {
     static COLORREF crColor[8] = { RGB (0,     0, 0), RGB (  0,   0, 255),
                                    RGB (0,   255, 0), RGB (  0, 255, 255),
                                    RGB (255,   0, 0), RGB (255,   0, 255),
                                    RGB (255, 255, 0), RGB (255, 255, 255) } ;
     HBRUSH          hBrush ;
     HDC             hdc ;
     RECT            rect ;
```

```
     hdc = GetDC (hwnd) ;
     GetClientRect (hwnd, &rect) ;
     hBrush = CreateSolidBrush (crColor[iColor - IDD_BLACK]) ;
     hBrush = (HBRUSH) SelectObject (hdc, hBrush) ;

     if (iFigure == IDD_RECT)
          Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom) ;
     else
          Ellipse   (hdc,. rect.left, rect.top, rect.right, rect.bottom) ;

     DeleteObject (SelectObject (hdc, hBrush)) ;
     ReleaseDC (hwnd, hdc) ;
     }

void PaintTheBlock (HWND hCtrl, int iColor, int iFigure)
     {
     InvalidateRect (hCtrl, NULL, TRUE) ;
     UpdateWindow (hCtrl) ;
     PaintWindow (hCtrl, iColor, iFigure) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE hInstance ;
     PAINTSTRUCT      ps ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_ABOUT :
                         if (DialogBox (hInstance,   boutBox", hwnd,
                                        AboutDlgProc))
                              InvalidateRect (hwnd, NULL, TRUE) ;
                         return 0 ;
                    }
               break ;

          case WM_PAINT :
               BeginPaint (hwnd, &ps) ;
               EndPaint (hwnd, &ps) ;

               PaintWindow (hwnd, iCurrentColor, iCurrentFigure) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
```

```
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam,
                                                          LPARAM lParam)
     {
     static HWND hCtrlBlock ;
     static int  iColor, iFigure ;

     switch (iMsg)
          {
          case WM_INITDIALOG :
               iColor  = iCurrentColor ;
               iFigure = iCurrentFigure ;

               CheckRadioButton (hDlg, IDD_BLACK, IDD_WHITE, iColor) ;
               CheckRadioButton (hDlg, IDD_RECT,  IDD_ELL,   iFigure) ;

               hCtrlBlock = GetDlgItem (hDlg, IDD_PAINT) ;

               SetFocus (GetDlgItem (hDlg, iColor)) ;
               return FALSE ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDOK :
                         iCurrentColor  = iColor ;
                         iCurrentFigure = iFigure ;
                         EndDialog (hDlg, TRUE) ;
                         return TRUE ;

                    case IDCANCEL :
                         EndDialog (hDlg, FALSE) ;
                         return TRUE ;

                    case IDD_BLACK :
                    case IDD_RED :
                    case IDD_GREEN :
                    case IDD_YELLOW :
                    case IDD_BLUE :
                    case IDD_MAGENTA :
                    case IDD_CYAN :
                    case IDD_WHITE :
                         iColor = LOWORD (wParam) ;
                         CheckRadioButton (hDlg, IDD_BLACK, IDD_WHITE,
                                           LOWORD (wParam)) ;

                         PaintTheBlock (htrlBlock, iColor, iFigure) ;
                         return TRUE ;

                    case IDD_RECT :
```

```
                    case IDD_ELL :
                         iFigure = wParam ;
                         CheckRadioButton (hDlg, IDD_RECT, IDD_ELL,
                                           LOWORD (wParam)) ;
                         PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
                         return TRUE ;
                    }
               break ;

          case WM_PAINT :
               PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
               break ;
          }
     return FALSE ;
     }
```

ABOUT2.RC

```
/*----------------------------
   ABOUT2.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about2.h"

about2 ICON about2.ico

About2 MENU
     {
     POPUP "&Help"
          {
          MENUITEM "&About About2...",       IDM_ABOUT
          }
     }

#define TABGRP (WS_TABSTOP | WS_GROUP)

AboutBox DIALOG 20, 20, 140, 188
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT       "About2"       -1,            0,  12, 140,   8
     ICON        "About2"       -1,            8,   8,   0,   0
     CTEXT       "About Box Demo Program" -1, 4,  36, 130,   8
     CTEXT       ""             IDD_PAINT,    68,  54,  60,  60
     GROUPBOX    "&Color"       -1,            4,  50,  54, 112
     RADIOBUTTON "&Black"       IDD_BLACK,     8,  60,  40,  12, TABGRP
     RADIOBUTTON "B&lue"        IDD_BLUE,      8,  72,  40,  12
     RADIOBUTTON "&Green"       IDD_GREEN,     8,  84,  40,  12
     RADIOBUTTON "Cya&n"        IDD_CYAN,      8,  96,  40,  12
     RADIOBUTTON "&Red"         IDD_RED,       8, 108,  40,  12
     RADIOBUTTON "&Magenta"     IDD_MAGENTA,   8, 120,  40,  12
     RADIOBUTTON "&Yellow"      IDD_YELLOW,    8, 132,  40,  12
     RADIOBUTTON "&White"       IDD_WHITE,     8, 144,  40,  12
     GROUPBOX    "&Figure"      -1,           68, 120,  60,  40, WS_GROUP
```

```
RADIOBUTTON "Rec&tangle"   IDD_RECT,     72, 134,  50,  12, TABGRP
RADIOBUTTON "&Ellipse"     IDD_ELL,      72, 146,  50,  12
DEFPUSHBUTTON "OK"         IDOK,         20, 168,  40,  14, WS_GROUP
PUSHBUTTON  "Cancel"       IDCANCEL,     80, 168,  40,  14, WS_GROUP
}
```

ABOUT2.H

```
/*----------------------
   ABOUT2.H header file
  ----------------------*/

#define IDM_ABOUT       1

#define IDD_BLACK      10
#define IDD_BLUE       11
#define IDD_GREEN      12
#define IDD_CYAN       13
#define IDD_RED        14
#define IDD_MAGENTA    15
#define IDD_YELLOW     16
#define IDD_WHITE      17

#define IDD_RECT       20
#define IDD_ELL        21

#define IDD_PAINT      30
```

ABOUT2.ICO

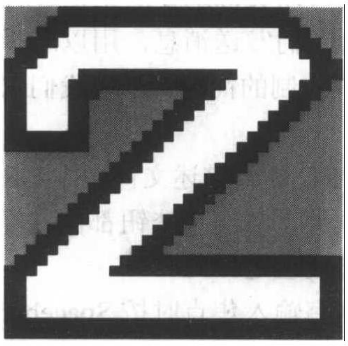


图 11-3 ABOUT2程序

ABOUT2 中的 About 框有两组单选钮。一组用来选择颜色，另一组用来选择是矩形还是椭圆。所选的矩形或者椭圆显示在对话框内，其内部以当前选择的颜色着色。用户按下 OK 按钮后，对话框终止，程序的窗口过程在它自己的客户区内绘出所选图形。如果你按下 Cancel，则主窗口的客户区保持原样。对话框如图 11-4 所示。尽管 ABOUT2 使用预定义的标识符 IDOK 和 IDCANCEL 作为两个压入按钮，但是每个单选钮均有自己的标识符，它们以字母 IDD 打头（“ ID ”代表对话框控制）。这些标识符在 ABOUT2.H 中定义。

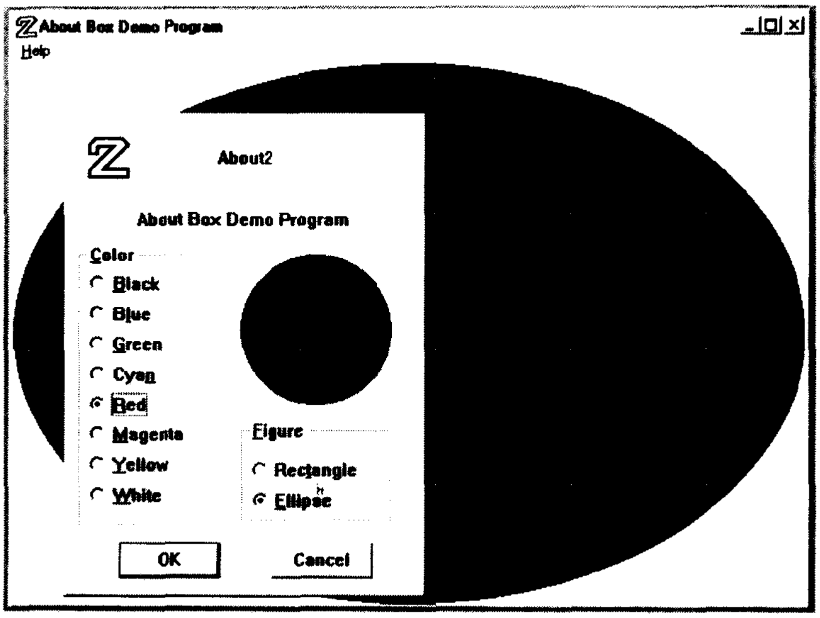


图 11-4 ABOUT2程序的对话框

11.1.8 使用对话框控制

在第 8 章中，你发现大多数子窗口控制给其父窗口发送 WM_COMMAND 消息（唯一的例外是滚动条控制）。你还看到，通过给子窗口控制发消息，父窗口可以改变子窗口控制（例如，选择或者不选择单选钮或者复选框）。你也可以类似地在对话框过程中改变控制。例如，如果你设计了一系列单选钮，就可以通过给它们发送消息，用以选择或者不选择这些按钮。不过， Windows 也提供了几种使用对话框控制的简捷方法。我们来看一看对话框过程与子窗口控制相互通信的方式。

ABOUT2 的对话框模板显示在图 11-3 的 ABOUT2.RC 资源描述文件中。 GROUPBOX 控制只是一个带标题（标题为 Color 或者 Figure ）的框架，每组单选钮都由这样的框架包围。前一组的八个单选钮是互斥的，第二组的两个单选钮也是如此。

当用鼠标单击其中一个单选钮时（或者当单选钮拥有输入焦点时按 Spacebar ），子窗口向其父窗口发送一条 WM_COMMAND 消息，消息的 wParam 的低位字被置为控制的 ID ，wParam 的高位字是一个通知码， lParam 值是控制的窗口句柄。对单选钮，这个通知码是 BN_CLICKED 或者 0 。然后 Windows 中的对话框窗口过程将这个 WM_COMMAND 消息发送给 ABOUT2.C 内的对话框过程。当对话框过程收到一条 WM_COMMAND 消息时，它为此按钮设置选中标记，并为组中其它按钮清除选中标记。

你可能还记得在第 8 章中已经讲过，选中和不选中按钮均需要向子窗口控制发送 BM_CHECK 消息。要设置一个按钮选中标记，你可以使用：

```
SendMessage (hwndCtrl, BM_SETCHECK, 1, 0) ;
```

要清除选中标记，你可以使用：

```
SendMessage (hwndCtrl, BM_SETCHECK, 0, 0) ;
```

其中， hwndCtrl 是子窗口按钮控制的窗口句柄。

但是在对话框过程中使用这种方法时有点问题，因为你不知道所有单选钮的窗口句柄，只是从你获得的消息中知道其中一个句柄。幸运的是， Windows 为你提供了一个函数，可以用对话框句柄和控制 ID 来获取一个对话框控制的窗口句柄：

```
hwndCtrl = GetDlgItem (hDlg, id) ;
```

（你也可以使用如下函数从窗口句柄中获取控制的 ID 值：

```
id = GetWindowLong (hwndCtrl, GWL_ID) ;
```

但是在多数情况下这是不必要的。）

你会注意到，在图 11-3 所示的头文件 ABOUT2.H 中，八种颜色的 ID 值是从 IDD_BLACK 到 IDD_WHITE 连续变化的，这种安排在处理来自单选钮的 WM_COMMAND 消息时将很有用。在第一次尝试选中或者不选中单选钮时，你可能会在对话框过程中编写如下的程序：

```
static int iColor ;
[other program lines]
case WM_COMMAND :
     switch (LOWORD (wParam))
          {
          [other program lines]
          case IDD_BLACK :
          case IDD_RED :
          case IDD_GREEN :
          case IDD_YELLOW :
          case IDD_BLUE :
          case IDD_MAGENTA :
          case IDD_CYAN :
          case IDD_WHITE :
               iColor = LOWORD (wParam) ;

               for (i = IDD_BLACK ; i <= IDD_WHITE ; i++)
                    SendMessage (GetDlgItem (hDlg, i),
                         BM_SETCHECK, i == LOWORD (wParam), 0) ;
               return TRUE ;
          [other program lines]
```

这种方法能令人满意地运行。你将新的颜色值保存在 iColor 中，并且还建立了一个循环，轮流使用所有八种颜色的 ID 值。你获取每个单选钮控制的窗口句柄，并用 SendMessage 给每个句柄发一条 BM_SETCHECK 消息。只有对于向对话框窗口过程发送 WM_COMMAND 消息的按钮，这条消息的 wParam 值才被设置为 1 。

第一种简化的方法是使用专门的对话框过程 SendDlgItemMessage:

```
SendDlgItemMessage (hDlg, id, iMsg, wParam, lParam) ;
```

它等价于：

```
SendMessage (GetDlgItem (hDlg, id), id, wParam, lParam) ;
```

现在，循环将形如：

```
for (i=IDD_BLACK; i<IDD_WHITE; i++)
    SendDlgItemMessage (hDlg, i, BM_SETCHECK, i==LOWORD (wParam), 0) ;
```

稍微有些改进。但是真正的重大突破要等到使用了 CheckRadioButton 函数时才会出现：

```
CheckRadioButton (hDlg, idFirst, idLast, idCheck) ;
```

这个函数将 ID 在 idFirst 到 idLast 之间的所有单选钮的选中标记都清除掉，除了 ID 为 idCheck 的单选钮，因为它是被选中的。这里，所有 ID 必须是连续的。从而我们可以完全摆脱循环，并使用：

```
CheckRadioButton (hDlg, IDD_BLACK, IDD_WHITE, LOWORD (wParam)) ;
```

这正是 ABOUT2 对话框过程所采用的方法。

在使用复选框时，也提供了类似的简化函数。如果你创建了一个 CHECKBOX 对话窗口控制，那么可以使用如下函数来设置和清除选中标记：

```
CheckDlgButton (hDlg, idCheckbox, iCheck) ;
```

如果 iCheck 设置为 1，那么按钮被选中；如果设置为 0，那么按钮不被选中。你可以使用如下方法来获取对话框中某个复选框的状态：

```
iCheck = IsDlgButtonChecked (hDlg, idCheckbox) ;
```

在对话框过程中，你既可以将选中标记的当前状态保存在一个静态变量中，又可以在收到一条 WM_COMMAND 消息后，使用如下方法触发按钮：

```
CheckDlgButton (hDlg, idCheckbox,
    !IsDlgButtonChecked (hDlg, idCheckbox)) ;
```

如果你定义了 BS_AUTOCHECKBOX 控制，那么完全没有必要处理 WM_COMMAND 消息。在终止对话框前，你只要使用 IsDlgButtonChecked 就可以获取按钮的当前状态。

11.1.9 OK 和 Cancel 按钮

ABOUT2 有两个压入按钮，分别标记为 OK 和 Cancel。在 ABOUT2.RC 的对话框模板中，OK 按钮的 ID 值为 IDOK（在 Windows 头文件中被定义为 1），Cancel 按钮的 ID 值为 IDCANCEL（在 Windows 头文件中被定义为 0），OK 按钮是缺省的：

```
DEFPUSHBUTTON "OK"     IDOK,       20, 168, 40, 14, WS_GROUP
PUSHBUTTON    "Cancel" IDCANCEL,   80, 168, 40, 14, WS_GROUP
```

在对话框中，通常都这样安排 OK 和 Cancel 按钮：将 OK 按钮作为缺省按钮有助于用键盘界面终止对话。通常情况下，你通过单击两个鼠标按钮之一或者当所期望的按钮具有输入焦点时按下 Spacebar 来终止对话框。不过，如果用户按下 Enter，对话框窗口过程也将产生一条 WM_COMMAND 消息，而不管哪个控制具有输入焦点。wParam 的低位字被

设置为对话框中缺省压入按钮的 ID 值，除非另一个压入按钮拥有输入焦点。在后一种情况下，wParam 的低位字被设置为具有输入焦点的压入按钮的 ID 值。如果对话框中没有缺省压入按钮，那么 Windows 向对话框过程发送一条 WM_COMMAND 消息，消息中 wParam 的低位字被设置为 IDOK。如果用户按下 Esc 键或者 Ctrl-Break 键，那么 Windows 令 wParam 等于 IDCANCEL，并给对话框过程发送一条 WM_COMMAND 消息。所以，你不用在对话框过程中加入单独的处理键盘的操作，因为通常终止对话框的击键会由 Windows 将这两个压入按钮动作转换为 WM_COMMAND 消息。

AboutDlgProc 函数通过调用 EndDialog 来处理这两种 WM_COMAND 消息：

```
switch (LOWORD (wParam))
     {
     case IDOK :
          iCurrentColor  = iColor ;
          iCurrentFigure = iFigure ;
          EndDialog (hDlg, TRUE) ;
          return TRUE ;

     case IDCANCEL :
          EndDialog (hDlg, FALSE) ;
          return TRUE ;
```

ABOUT2 的窗口过程在程序的客户区中绘制矩形或椭圆时使用了全局变量 iCurrentColor 和 iCurrentFigure。AboutDlgProc 在对话框中画图时使用了静态局部变量 iColor 和 iFigure。

注意 EndDialog 的第二个参数的不同值，这个值是在 WndProc 中作为原 DialogBox 函数的返回值传回的：

```
case IDM_ABOUT:
    if (DialogBox (hInstance, "AboutBox", hwnd, AboutDlgProc))
        InvalidateRect (hwnd, NULL, TRUE) ;
    return 0;
```

如果 DialogBox 返回 TRUE（非零），则意味着按下了 OK 按钮，然后需要使用新的颜色来刷新 WndProc 客户区。当 AboutDlgProc 收到一条 WM_COMMAND 消息，并且消息的 wParam 的低位字等于 IDOK 时，AboutDlgProc 将图形和颜色保存在全局变量 iCurrentColor 和 iCurrentFigure 中。如果 DialogBox 返回 FALSE，则主窗口继续使用 iCurrentColor 和 iCurrentFigure 的原始设置。

TRUE 和 FALSE 通常用于 EndDialog 调用中，以告知主窗口过程用户是用 OK 还是用 Cancel 来终止对话框的。不过，EndDialog 的参数实际上是一个 int 值，而 DialogBox 也返回一个 int 值，所以，用这种方法能比仅用 TRUE 或者 FALSE 返回更多的信息。

11.1.10 Tab 停留位和组

在第 8 章中，我们利用窗口子类化为 COLORS1 增加功能，使我们能够通过按 Tab 键

从一个滚动条转移到另一个滚动条。在对话框中，窗口子类化是不必要的，因为 Windows 完成了将输入焦点从一个控制移动到另一个控制的所有工作。尽管如此，你必须在对话框模板中使用 WS_TABSTOP 和 WS_GROUP 窗口风格达到此目的。对于所有想要使用 Tab 键访问的控制，要在其窗口风格中指定 WS_TABSTOP 。如果你往回参阅“关于定义控制的补充说明”那一节的表格，会注意到许多控制将 WS_TABSTOP 定义为缺省风格，其它一些则没有将它作为缺省风格。一般而言，不包含 WS_TABSTOP 风格的控制（特别是静态控制）不应该获取输入焦点，因为即使有了输入焦点，它们也不能完成操作。除非在处理 WM_INITDIALOG 消息时你将输入焦点设置给一个特定的控制，并从消息中返回 FALSE；否则 Windows 将输入焦点设置为对话框内第一个具有 WS_TABSTOP 风格的控制。

Windows 给对话框增加的第二个键盘接口包括光标移动键，这种界面对于单选钮有特殊的重要性。如果你使用 Tab 键移动到某一组内当前选中的单选钮，那么，就需要使用光标移动键将输入焦点从该单选钮移动到组内其它单选钮上。使用 WS_GROUP 窗口风格即可获得这个功能。对于对话框模板中的特定控制序列， Windows 将使用光标移动键把输入焦点从第一个具有 WS_GROUP 风格的控制切换到下一个具有 WS_GROUP 风格的控制中。如果有必要， Windows 将从对话框的最后一个控制循环到第一个控制，以便找到组的结尾。

在缺省设置下，控制 LTEXT 、 CTEXT 、 RTEXT 和 ICON 包含有 WS_GROUP 风格，这种风格方便地标记了组的结尾。你必须经常将 WS_GROUP 风格加到其它类型的控制中。

让我们来看一看 ABOUT2.RC 中的对话框模板：

```
AboutBox DIALOG 20, 20, 140, 188
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT       "About2"      -1,          0,  12, 140,   8
     ICON        "About2"      -1,          8,   8,   0,   0
     CTEXT       "About Box Demo Program" -1, 4,  36, 130,   8
     CTEXT       ""            IDD_PAINT,  68,  54,  60,  60
     GROUPBOX    "&Color"      -1,          4,  50,  54, 112
     RADIOBUTTON "&Black"      IDD_BLACK,   8,  60,  40,  12, TABGRP
     RADIOBUTTON "B&lue"       IDD_BLUE,    8,  72,  40,  12
     RADIOBUTTON "&Green"      IDD_GREEN,   8,  84,  40,  12
     RADIOBUTTON "Cya&n"       IDD_CYAN,    8,  96,  40,  12
     RADIOBUTTON "&Red"        IDD_RED,     8, 108,  40,  12
     RADIOBUTTON "&Magenta"    IDD_MAGENTA, 8, 120,  40,  12
     RADIOBUTTON "&Yellow"     IDD_YELLOW,  8, 132,  40,  12
     RADIOBUTTON "&White"      IDD_WHITE,   8, 144,  40,  12
     GROUPBOX    "&Figure"     -1,         68, 120,  60,  40, WS_GROUP
     RADIOBUTTON "Rec&tangle"  IDD_RECT,   72, 134,  50,  12, TABGRP
     RADIOBUTTON "&Ellipse"    IDD_ELL,    72, 146,  50,  12
     DEFPUSHBUTTON "OK"        IDOK,       20, 168,  40,  14, WS_GROUP
     PUSHBUTTON  "Cancel"      IDCANCEL,   80, 168,  40,  14, WS_GROUP
     }
```

为了简化模板的写法，在 ABOUT2.RC 中定义了一个标识符以组合 WS_TABSTOP 和

WS_GROUP：

```
#define TABGRP  (WS_TABSTOP | WS_GROUP)
```

四个具有 WS_TABSTOP 风格的控制是每个组的第一个单选钮（显式地包含）和两个压入按钮（缺省设置）。在第一次激活对话框时，你可以使用 Tab 键在这四个控制之间移动。

在每组单选钮中，你可以使用光标移动键切换输入焦点并改变选中标记。例如，Color 分组框的第一个单选钮和 Figure 分组框都具有 WS_GROUP 风格。这意味着你可以用光标移动键将焦点从 Black 单选钮移动到 Figure 分组框中。类似地，Figure 分组框的第一个单选钮 (Rectangle)和 DEFPUSHBUTTON 都具有 WS_GROUP 风格，所以你可以使用光标移动键在组内两个单选钮：Rectangle 和 Ellipse 之间移动。两个压入按钮都有 WS_GROUP 风格，以阻止光标移动键在压入按钮具有输入焦点时起作用。

使用 ABOUT2 时，Windows 的对话框管理器在两组单选钮中完成一些相当复杂的处理。正如所期望的那样，处于单选钮组内时，光标移动键切换输入焦点，并给对话框过程发送 WM_COMMAND 消息。但是，当你改变了组内的选中的单选钮时，Windows 也给新选中的单选钮赋予 WS_TABSTOP 风格。当你下一次使用 Tab 切换到这一组后，Windows 将会把输入焦点设置为选中的单选钮。

文本字段中的“&”导致紧跟其后的字母以下划线显示，这就增加了另一种键盘接口，你可以通过按下划线字母来将输入焦点移动到任意单选钮上。通过按 C（代表 Color 分组框）或者 F（代表 Figure 分组框），你可以将输入焦点移动到相应组内当前选中的单选钮上。

尽管程序员通常让对话框管理器来完成这些工作，但是 Windows 提供了两个函数，以便程序员查找下一个或者前一个 Tab 键停留项或者组项。这些函数为：

```
hwndCtrl = GetNextDlgTabItem (hDlg, hwndCtrl, bPrevious) ;
```

和：

```
hwndCtrl = GetNextDlgGroupItem (hDlg, hwndCtrl, bPrevious) ;
```

如果 bPrevious 为 TRUE，那么函数返回前一个 Tab 键停留项或组项；如果为 FALSE，则返回下一个 Tab 键停留项或者组项。

11.1.11 在对话框上画图

ABOUT2 还完成了一些相对而言不同寻常的事情，即在对话框上画图。让我们来看一看它是怎样做的。在 ABOUT2.RC 的对话框模板内，使用位置和大小为我们想要画图的区域定义了一块空白文本控制：

```
CTEXT "" IDD_PAINT, 68, 54, 60, 60
```

这个区域为 15 个字符宽和 7.5 个字符高。由于这个控制没有文本，所以窗口过程为“静态”类所作的工作只是在必须重绘这个子窗口控制时清除其背景。

在当前颜色或者图形选择发生改变或者对话框自身获得一条 WM_PAINT 消息时，对话框过程调用 PaintTheBlock，这个函数在 ABOUT2.C 中：

```
PaintTheBlock (hCtrlBlock, iColor, iFigure) ;
```

窗口句柄 hCtrlBlock 已经在处理 WM_INITDIALOG 消息时被设置：

```
hCtrlBlock = GetDlgItem (hDlg, IDD_PAINT) ;
```

下面是 PaintTheBlock 函数：

```
void PaintTheBlock  (HWND hCtrl, int iColor, int iFigure)
    {
    InvalidRect (hCtrl, NULL, TRUE) ;
    UpdateWindow (hCtrl) ;
    PaintWindow (hCtrl, iColor, iFigure) ;
    }
```

这个函数使得子窗口控制无效，表示它已经被更新，并且调用 ABOUT2 中的另一个函数 PaintWindow 。

PaintWindow 函数获取一个设备环境句柄，并将其放到 hCtrl 中，画出所选图形，根据所选颜色用一个着色画刷填充图形。子窗口控制的大小从 GetClientRect 获得。尽管对话框模板以字符为单位定义了控制的大小，但 GetClientRect 获取以象素点为单位的尺寸。你也可以使用 MapDialogRect 函数将对话框中的字符坐标转换为客户区中的象素点坐标。

我们并非真的绘制了对话框的客户区，实际上绘制的是子窗口控制的客户区。每当对话框得到一条 WM_PAINT 消息时，就令子窗口控制失效，并刷新它，使它确信现在其客户区又有效了，然后再在其上画图。

11.1.12 将其它函数用于对话框

大多数可以用于子窗口的函数也可以用于对话框中的控制。例如，如果你想捣乱的话，那么可以使用 MoveWindow 在对话框内移动控制，强迫用户用鼠标来跟踪它们。

有时，你需要根据其它控制的设置动态地启用或者禁用某些控制，这需要调用：

```
EnableWindow (hwndCtrl, bEnable) ;
```

当 bEnable 为 TRUE （非零）时，它启用控制；当 bEnable 为 FALSE （ 0 ）时，它禁用控制。在控制被禁用时，它不再接收键盘或者鼠标输入。不能禁用一个拥有输入焦点的控制。

11.1.13 定义自己的控制

尽管 Windows 承担了许多维护对话框和子窗口控制的工作，但是它同时也给你提供了各种加入代码的方法。前面我们已经看到了在对话框上绘图的方法。你也可以使用第 8 章中讨论的窗口子类化来改变子窗口控制的操作。

你还可以定义自己的子窗口控制，并将它们用到对话框中。例如，假定你特别不喜欢普通的矩形压入按钮，而倾向于创建椭圆形压入按钮；那么你可以通过注册一个窗口类，并使用自己编制的窗口过程处理来自你所创建窗口的消息，从而创建椭圆形压入按钮；然后，你在对话框模板的 CONTROL 语句中指定这个窗口类。图 11-5 所示的 ABOUT3 程序

正是这样做的。

ABOUT3.MAK

```
#--------------------
# ABOUT3.MAK make file
#--------------------

about3.exe : about3.obj about3.res
     $(LINKER) $(GUIFLAGS) -OUT:about3.exe about3.obj about3.res $(GUILIBS)

about3.obj : about3.c about3.h
     $(CC) $(CFLAGS) about3.c

about3.res : about3.rc about3.h about3.ico
     $(RC) $(RCVARS) about3.rc
```

ABOUT3.C

```
/*------------------------------------------
   ABOUT3.C -- About Box Demo Program No. 3
               (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>
#include "about3.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK EllipPushWndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "About3" ;
     MSG          msg ;
     HWND         hwnd ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;
```

```
     wndclass.cbSize         = sizeof (wndclass) ;
     wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc    = EllipPushWndProc ;
     wndclass.cbClsExtra     = 0 ;
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = NULL ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = "EllipPush" ;
     wndclass.hIconSm       = NULL ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "About Box Demo Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE hInstance ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = ((LPCREATESTRUCT) lParam)->hInstance ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_ABOUT :
                         DialogBox (hInstance, "AboutBox", hwnd,
                                    AboutDlgProc) ;
                         return 0 ;
                    }
               break ;
```

```
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_INITDIALOG :
               return TRUE ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDOK :
                         EndDialog (hDlg, 0) ;
                         return TRUE ;
                    }
               break ;
          }
     return FALSE ;
     }

LRESULT CALLBACK EllipPushWndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                          LPARAM lParam)
     {
     char        szText[40] ;
     HBRUSH      hBrush ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_PAINT :
               GetClientRect (hwnd, &rect) ;
               GetWindowText (hwnd, szText, sizeof (szText)) ;

               hdc = BeginPaint (hwnd, &ps) ;

               hBrush = CreateSolidBrush (GetSysColor (COLOR_WINDOW)) ;
               hBrush = (HBRUSH) SelectObject (hdc, hBrush) ;
               SetBkColor (hdc, GetSysColor (COLOR_WINDOW)) ;
               SetTextColor (hdc, GetSysColor (COLOR_WINDOWTEXT)) ;

               Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom) ;
               DrawText (hdc, szText, -1, &rect,
                              DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

               DeleteObject (SelectObject (hdc, hBrush)) ;
```

```
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_KEYUP :
          if (wParam != VK_SPACE)
               break ;
                                        // fall through
     case WM_LBUTTONUP :
          SendMessage (GetParent (hwnd), WM_COMMAND,
               GetWindowLong (hwnd, GWL_ID), (LPARAM) hwnd) ;
          return 0 ;
     }
return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

ABOUT3.RC

```
/*---------------------------
   ABOUT3.RC resource script
  ---------------------------*/

#include <windows.h>
#include "about3.h"

about3 ICON about3.ico

About3 MENU
     {
     POPUP "&Help"
          {
          MENUITEM "&About About3...",        IDM_ABOUT
          }
     }

#define TABGRP (WS_TABSTOP | WS_GROUP)

AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT   "About3"                         -1,  0, 12, 160,  8
     ICON    "About3"                         -1,  8,  8,   0,  0
     CTEXT   "About Box Demo Program"         -1,  0, 36, 160,  8
     CTEXT   "(c) Charles Petzold, 1996"      -1,  0, 48, 160,  8
     CONTROL "OK" IDOK, "EllipPush",    TABGRP,    64, 60,  32, 14
     }
```

ABOUT3.H

```
/*----------------------
   ABOUT3.H header file
  ----------------------*/

#define IDM_ABOUT       1
```

ABOUT3.ICO

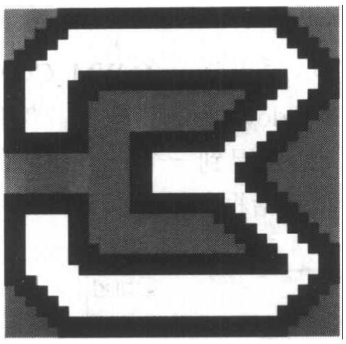


图 11-5 ABOUT3程序

我们所注册的窗口类叫做 EllipPush（“椭圆压入按钮”）。在对话框模板中，我们没有使用 DEFPUSHBUTTON 语句，而是使用了一个 CONTROL 语句来指定这个窗口类：

```
CONTROL "OK" IDOK, "EllipPush", TABGRP, 64, 60, 32, 14
```

当在对话框中创建子窗口控制时，对话框管理器把这个窗口类用于 CreateWindow 调用中。

ABOUT3.C 程序在 WinMain 中注册了"EllipPush"窗口类：

```
wndclass.cbSize        = sizeof (wndclass) ;
wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
wndclass.lpfnWndProc   = EllipPushWndProc ;
wndclass.cbClsExtra    = 0 ;
wndclass.cbWndExtra    = 0 ;
wndclass.hInstance     = hInstance ;
wndclass.hIcon         = NULL ;
wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground = (HBRUSH) (COLOR_BTNFACE + 1) ;
wndclass.lpszMenuName  = NULL ;
wndclass.lpszClassName = "EllipPush" ;
wndclass.hIconSm       = NULL ;

RegisterClassEx (&wndclass) ;
```

该窗口类指定窗口过程为 EllipPushWndProc 。

EllipPushWndProc 窗口过程只处理三种消息： WM_PAINT 、 WM_KEYUP 和 WM_LBUTTONUP 。在处理 WM_PAINT 消息时，它从 GetClientRect 中获取窗口的大小，从 GetWindowText 中获取显示在压入按钮上的文本，用 Windows 函数 Ellipse 和 DrawText 来输出椭圆和文本。

WM_KEYUP 和 WM_LBUTTONUP 消息的处理非常简单：

```
case WM_KEYUP :
     if (wParam != VK_SPACE)
          break ;
                              // fall through
case WM_LBUTTONUP :
     SendMessage (GetParent (hwnd), WM_COMMAND,
```

```
        GetWindowLong (hwnd, GWL_ID), (LPARAM) hwnd) ;
    return 0 ;
```

窗口过程使用 GetParent 来获取其父窗口(即对话框)的句柄，并发送一条 WM_COMMAND 消息，消息的 wParam 等于控制的 ID，这个 ID 是用 GetWindowLong 获取的。然后，对话框窗口过程将这条消息传给 ABOUT3 内的对话框过程，结果得到一个用户定制的压入按钮，如图 11-6 所示。你可以用同样的方法来创建其它定制对话框控制。

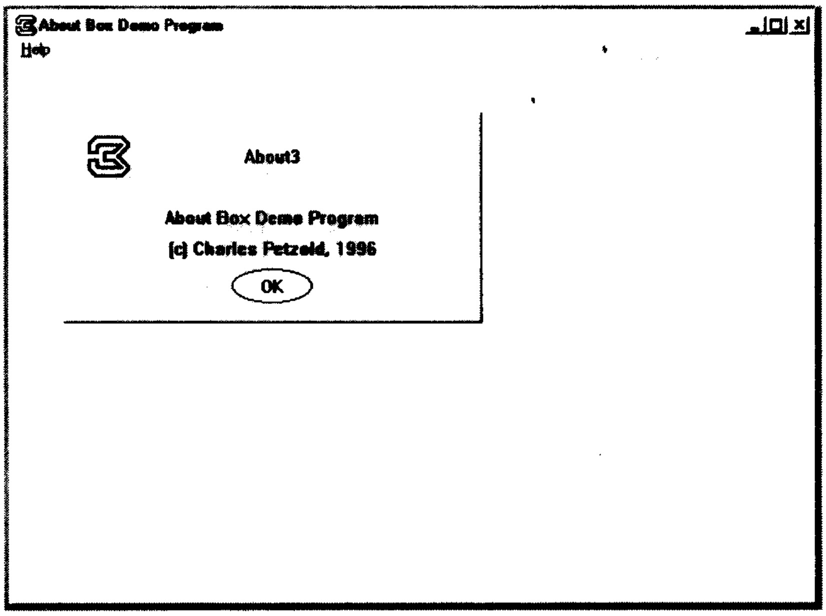


图 11-6 ABOUT3创建的定制压入按钮

这就是全部要做的吗？其实不然。对于通常维护子窗口控制所需要的处理而言，EllipPushWndProc 只是一个空架子。例如，在 EllipPushWndProc 中，按钮不会象普通的压入按钮那样闪烁。要翻转压入按钮内的颜色，窗口过程必须处理 WM_KEYDOWN （来自 Spacebar ）和 WM_LBUTTONDOWN 消息。窗口过程还必须在收到 WM_LBUTTONDOWN 消息时捕获鼠标，并且，如果当按钮还处于按下状态时，鼠标移到了子窗口的客户区之外，那么要释放鼠标捕获（并将按钮的内部颜色返回为正常状态）。只有在鼠标被捕获时松开按钮，子窗口才会给其父窗口送回一条 WM_COMMAND 消息。

EllipPushWndProc 也不处理 WM_ENABLE 消息。如上所述，对话框过程可以使用 EnableWindow 函数来禁用某窗口，于是，子窗口将显示灰色文本，而不再是黑色文本，以表示它已经被禁用，而不能再接收任何消息了。

如果子窗口控制的窗口过程需要为所创建的每个窗口存放各自不同的数据，那么它可以通过使用窗口类结构中的 cbWndExtra 值来做到。这样就在内部窗口结构中保留了空间，并可以用 SetWindowWord 、 SetWindowLong 、 GetWindowWord 和 GetWindowLong 来访问该数据。

11.2 消息框

我们已经看到了定制对话框的方法。现在，让我们来看一看对话框的变体——消息框。在第 7 章中，我们就开始使用消息框了，但是还没有详细说明过它。

当你想从用户那里获得一个简单的应答时，使用消息框来代替对话框是很合适的，也是很容易的。一般的语法为：

```
iItem = MessageBox (hwndParent, szText, szCaption, iType) ;
```

消息框具有一条标题（字符串 szCaption）、一行或者多行文本（szText）、一个或者多个按钮和一个预定义的图标（可选），按钮中有一个是缺省的，MessageBox 返回的 iItem 的值指示被点中的按钮是哪一个。

hwndParent 参数一般是创建消息框的窗口的句柄。在清除消息框之后，输入焦点必须设置为这个窗口。如果你不知道窗口句柄，或者不希望输入焦点回到你的某个窗口，那么你可以将句柄置为 NULL。如果你在对话框中使用了消息框，那么该参数应该设置为对话框窗口句柄（我们称为 hDlg）。

szText 参数是一个长指针，指向显示在消息框中的以 NULL 结尾的文本。如果需要，Windows 将这个文本分为几行。文本中也可以包含制表符，你还可以使用回车、换行或者回车加换行来自己分行。szCaption 串通常是应用程序的名字。

iType 参数是用 C 的“或”运算符联合起来的标志集合。第一组标志位指定在消息框下部出现的压入按钮：MB_OK（缺省按钮）、MB_OKCANCEL、MB_YESNO、MB_YESNOCANCEL、MB_RETRYCANCEL、MB_ABORTRETRYIGNORE 和 MB_HELP。你可以看到，这些标志最多允许有四个按钮。第二组标志指定哪个按钮是缺省按钮：MB_DEFBUTTON1 (缺省设置)、MB_DEFBUTTON2、MB_DEFBUTTON3 和 MB_DEFBUTTON4。

第三组标志指定出现在消息框中的图标：MB_ICONINFORMATION(与 MB_ICONASTERISK 一样)、MB_ICONWARNING(和 MB_ICONEXCLAMATION 一样)、MB_ICONERROR(和 MB_ICONSTOP 及 MB_ICONHAND 一样)和 MB_ICONQUESTION。这里没有缺省图标。如果你没有使用这些标志，消息框就没有图标。你可以使用信息图标作为状态信息，感叹号作为提示，问号用作对行动后果的警告，停止图标作为严重问题的信号。

第四组标志决定消息框是应用程序模态的、还是系统模态的，在前一种情况下，你不用终止消息框就可以切换到另一个应用程序中，而在后一种情况下，在完成任何其它操作之前，你必须首先结束对话框。这两个标志为 MB_APPLMODAL（缺省值）和 MB_SYSTEMMODAL。最后，你还可以使用第五种标志 MB_NOFOCUS，这种情况下，消息框虽然显示但是没有输入焦点。

根据选中的是哪个按钮，消息框相应地返回下列标识符之一：IDOK、IDCANCEL、IDYES、IDNO、IDRETRY 和 IDABOUT。

11.2.1 弹出信息

在开发程序的过程中，消息框的一个方便的用途是在程序执行时为你提供信息。如果你能够像在 C 程序中使用 printf 语句那样频繁地使用消息对话框，那当然是很理想的，printf 语句只需要一个格式串和一些可变数目的变量。实际上，你可以创建一个函数，使你能够做到这一点：

```
void OkMsgBox (char *szCaption, char *szFormat, ...)
     {
     char  szBuffer[256] ;
     char *pArguments ;

     pArguments = (char *) &szFormat + sizeof (szFormat) ;
     vsprintf (szBuffer, szFormat, pArguments) ;
     MessageBox (NULL, szBuffer, szCaption, MB_OK) ;
     }
```

vsprintf 函数与 sprintf 相似，只是前者使用了指向一串参数的指针（ pArguments ），而不是参数本身。在调用 OkMsgBox 时，它将 pArguments 设置为堆栈上的参数。 OkMsgBox 的第一个参数是消息框的标题，第二个参数是一个格式串，第三个以及后面的参数是要显示的值。假定你想在每当窗口过程获得 WM_SIZE 消息时显示消息框，那么你可以编制如下代码：

```
case WM_SIZE :
     OkMsgBox ("WM_SIZE Message",
               "wParam = %04X-%04X, lParam = %04X-%04X",
               HIWORD (wParam), LOWORD (wParam),
               HIWORD (lParam), LOWORD (lParam)) ;
     [other program lines]
     return 0 ;
```

这段程序在消息框中显示 wParam 和 lParam 的值。

11.3 非模态对话框

在本章的开始，我曾经说过对话框分为“模态的”和“非模态的”两种。现在我们已经研究过了这两种对话框中最常见的一种——模态对话框。模态对话框（不包括系统模态对话框）允许用户在对话框与其它程序之间进行切换。但是，用户不能切换到同一程序的另一个窗口，直到模态对话框被清除为止。非模态对话框允许用户在对话框与其它程序之间进行切换，又可以在对话框与创建对话框的窗口之间进行切换。因此，非模态对话框与用户程序常见的普通弹出式窗口可能更相似。

当用户觉得让对话框保留片刻更加方便时，使用非模态对话框是合适的。例如，字处

理器经常使用非模态对话框来作 Find 和 Replace 对话。如果 Find 对话框是模态的，那么用户必须从菜单中选择 Find，输入要寻找的串，结束对话框，返回到文件中，然后再重复整个过程来寻找同一串的另一次出现。允许用户在文件与对话框之间进行切换则会方便得多。

你已经看到，模态对话框是使用 DialogBox 来创建的。只有在清除对话框之后，函数才返回值。在对话框过程内使用 EndDialog 调用来终止对话框，DialogBox 返回的是该调用的第二个参数的值。非模态对话框是使用 CreateDialog 来创建的，该函数所使用的参数与 DialogBox 相同。

```
hDlgModeless = CreateDialog (hInstance, szTemplate, hwndParent,
                             DialogProc) ;
```

区别是 CreateDialog 函数立即返回对话框的窗口句柄，通常将这个窗口句柄存放到全局变量中。

尽管将 DialogBox 这一名字用于模态对话框而 CreateDialog 用于非模态对话框是随意的，但是你可以通过非模态对话框与普通窗口类似这一点来记住这两个函数的区别。CreateDialog 可以令人想起 CreateWindow 函数来，后者创建的是普通窗口。

11.3.1 模态对话框与非模态对话框的区别

使用非模态对话框与使用模态对话框相似，但是也有一些重要的区别：

- 非模态对话框通常包含一个标题条和一个系统菜单框，在其对话框模板中的 STYLE 语句形如：

  ```
  STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
  ```

  标题条和系统菜单允许用户使用鼠标或者键盘将非模态对话框移动到另一个显示区域。对于模态对话框，你通常无需提供标题条和系统菜单，因为用户不能在其下面的窗口中做任何其它事情。

- 注意，在我们的示例 STYLE 语句中包含有 WS_VISIBLE 风格。如果忽略掉 WS_VISIBLE，那么你必须在调用 CreateDialog 之后调用 ShowWindow:

  ```
  hDlgModeless = CreateDialog (...) ;
  ShowWindow (hDlgModeless, SW_SHOW) ;
  ```

  如果你既没有包含 WS_VISIBLE 风格，又没有调用 ShowWindow，那么非模态对话框不会被显示。如果忽略这个事实，那么习惯于模态对话框的程序员在第一次试图创建非模态对话框时，将会经常出现问题。

- 与模态对话框和消息框的消息不同，非模态对话框的消息要经过程序的消息队列。要将这些消息传送给对话框窗口过程，则必须改变消息循环。方法如下：当你使用 CreateDialog 创建非模态对话框时，应该将从调用中返回的对话框句柄保存在一个全局变量（如 hDlgModelsess）中，并将消息循环改变为：

```
while (GetMessage (&msg, NULL, 0, 0))
        {
        if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
              {
              TranslateMessage (&msg) ;
              DispatchMessage  (&msg) ;
              }
        }
```

如果消息是发送给非模态对话框的，那么 IsDialogMessage 将它发送给对话框窗口过程，并返回 TRUE（非 0）；否则，它将返回 FALSE （0）。只有 hDlgModeless 为 0 或者消息不是该对话框的消息时，才必须调用 TranslateMessage 和 DispatchMessage 函数。如果你将键盘加速键用于你的程序窗口，那么消息循环将如下所示：

```
while (GetMessage (&msg, NULL, 0, 0))
        {
        if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
            {
            if (!TranslateAccelerator (hwnd, hAccel, &msg))
                {
                TranslateMessage (&msg) ;
                DispatchMessage  (&msg) ;
                }
            }
        }
```

由于全局变量被初始化为 0 ，所以 hDlgModeless 将为 0 ，直到创建对话框为止，从而保证了不会使用无效的窗口句柄来调用 IsDialogMessage 。在清除非模态对话框时，你也必须注意这一点，正如最后一点所说明的。

hDlgModeless 变量也可以由程序的其它部分使用，以便对非模态对话框是否存在加以验证。例如，程序中的其它窗口可以在 hDlgModeless 不等于 0 时给对话框发送消息。

- 使用 DestroyWindow 而不是 EndDialog 来结束非模态对话框。当你调用 DestroyWindow 后，将 hDlgModeless 全局变量设置为 0 。

用户习惯于从系统菜单中选择 Close 来结束非模态对话框。尽管启用了 Close 选项，Windows 内的对话框窗口过程并不处理 WM_CLOSE 消息。你必须自己在对话框过程中处理它：

```
case WM_CLOSE:
    DestroyWindow (hDlg) ;
    hDlgModeless = 0;
    break;
```

注意这两个窗口句柄之间的区别： DestroyWindow 的 hDlg 参数是传递给对话框过程的参

数；hDlgModeless 是从 CreateDialog 返回的全局变量，程序在消息循环内检验它。

你也可以允许用户使用压入按钮来关闭非模态对话框，使用的逻辑与处理 WM_CLOSE 消息一样。对话框必须返回给创建它的窗口的任何数据都可以被存储在全局变量中。

11.3.2 新的 COLORS 程序

第 8 章中所描述的 COLORS1 程序创建了九个子窗口，以便显示三个滚动条和六个文本项。那时候，这个程序还是我们所编制的程序中相当复杂的一个。如果将 COLORS1 转换为使用非模态对话框则会使程序——特别是其 WndProc 函数——变得令人难以置信的简单，修正后的 COLORS2 程序如图 11-7 所示。

COLORS2.MAK

```
#----------------------
# COLORS2.MAK make file
#----------------------

colors2.exe : colors2.obj colors2.res
     $(LINKER) $(GUIFLAGS) -OUT:colors2.exe colors2.obj colors2.res $(GUILIBS)

colors2.obj : colors2.c
     $(CC) $(CFLAGS) colors2.c

colors2.res : colors2.rc
     $(RC) $(RCVARS) colors2.rc
```

COLORS2.C

```
/*------------------------------------------------
   COLORS2.C - Version using Modeless Dialog Box
               (c) Charles Petzold, 1996
  ------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK ColorScrDlg  (HWND, UINT, WPARAM, LPARAM) ;

HWND hDlgModeless ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "Colors2" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
```

```
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = CreateSolidBrush (0L) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Color Scroll",
                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hDlgModeless = CreateDialog (hInstance, "ColorScrDlg", hwnd, ColorScrDlg) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          if (hDlgModeless == 0 || !IsDialogMessage (hDlgModeless, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage  (&msg) ;
               }
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_DESTROY :
               DeleteObject (
                    (HGDIOBJ) SetClassLong (hwnd, GCL_HBRBACKGROUND,
                         (LONG) GetStockObject (WHITE_BRUSH))) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK ColorScrDlg (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int iColor[3] ;
     HWND       hwndParent, hCtrl ;
     int        iCtrlID, iIndex ;
```

```
switch (iMsg)
     {
     case WM_INITDIALOG :
          for (iCtrlID = 10 ; iCtrlID < 13 ; iCtrlID++)
               {
               hCtrl = GetDlgItem (hDlg, iCtrlID) ;
               SetScrollRange (hCtrl, SB_CTL, 0, 255, FALSE) ;
               SetScrollPos   (hCtrl, SB_CTL, 0, FALSE) ;
               }
          return TRUE ;

     case WM_VSCROLL :
          hCtrl    = (HWND) lParam ;
          iCtrlID = GetWindowLong (hCtrl, GWL_ID) ;
          iIndex   = iCtrlID - 10 ;
          hwndParent = GetParent (hDlg) ;

          switch (LOWORD (wParam))
               {
               case SB_PAGEDOWN :
                    iColor[iIndex] += 15 ;        // fall through
               case SB_LINEDOWN :
                    iColor[iIndex] = min (255, iColor[iIndex] + 1) ;
                    break ;
               case SB_PAGEUP :
                    iColor[iIndex] -= 15 ;        // fall through
               case SB_LINEUP :
                    iColor[iIndex] = max (0, iColor[iIndex] - 1) ;
                    break ;
               case SB_TOP :
                    iColor[iIndex] = 0 ;
                    break ;
               case SB_BOTTOM :
                    iColor[iIndex] = 255 ;
                    break ;
               case SB_THUMBPOSITION :
               case SB_THUMBTRACK :
                    iColor[iIndex] = HIWORD (wParam) ;
                    break ;
               default :
                    return FALSE ;
               }
          SetScrollPos  (hCtrl, SB_CTL,      iColor[iIndex], TRUE) ;
          SetDlgItemInt (hDlg,  iCtrlID + 3, iColor[iIndex], FALSE) ;

          DeleteObject (
               (HGDIOBJ) SetClassLong (hwndParent, GCL_HBRBACKGROUND,
                    (LONG) CreateSolidBrush (
                         RGB (iColor[0], iColor[1], iColor[2])))) ;

          InvalidateRect (hwndParent, NULL, TRUE) ;
          return TRUE ;
     }
return FALSE ;
```

```
    }
```

COLORS2.RC

```
/*----------------------------
   COLORS2.RC resource script
  ----------------------------*/

#include <windows.h>

#define SBS_VERT_TAB (SBS_VERT | WS_TABSTOP)

ColorScrDlg DIALOG  8, 16, 124, 132
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
     CAPTION "Color Scroll Scrollbars"
     {
     CONTROL "&Red",    -1, "static",    SS_CENTER,    10,   4, 24,   8
     CONTROL "",        10, "scrollbar", SBS_VERT_TAB, 10,  16, 24, 100
     CONTROL "0",       13, "static",    SS_CENTER,    10, 120, 24,   8
     CONTROL "&Green",  -1, "static",    SS_CENTER,    50,   4, 24,   8
     CONTROL "",        11, "scrollbar", SBS_VERT_TAB, 50,  16, 24, 100
     CONTROL "0",       14, "static",    SS_CENTER,    50, 120, 24,   8
     CONTROL "&Blue",   -1, "static",    SS_CENTER,    90,   4, 24,   8
     CONTROL "",        12, "scrollbar", SBS_VERT_TAB, 90,  16, 24, 100
     CONTROL "0",       15, "static",    SS_CENTER,    90, 120, 24,   8
     }
```

图 11-7 COLORS2程序

原来的 COLORS1 程序所显示的滚动条大小是基于窗口大小的，而新程序在非模态对话框内以固定的尺寸来显示它们，如图 11-8 所示。

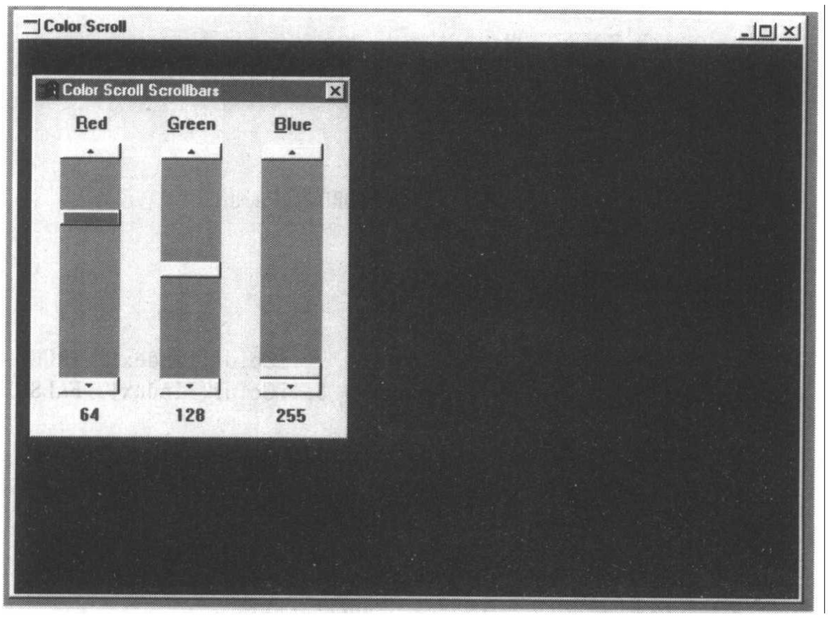


图 11-8 COLORS显示

COLORS2.RC 中的对话框模板对于对话框中的九个子窗口都使用 CONTROL 语句。在

COLORS2 中，非模态对话框是在 WinMain 函数里创建的，紧跟在用于程序的主窗口的 ShowWindow 调用之后。注意，主窗口的窗口风格包含 WS_CLIPCHILDREN，这允许程序无需擦除对话框就能够重画主窗口。

如上所述，从 CreateDialog 返回的对话框窗口句柄存放在全局变量 hDlgModeless 中，并在消息循环中被测试。不过，在这个程序中，不需要将句柄存放在全局变量中，也不需要在调用 IsDialogMessage 之前测试这个值。消息循环可以编写如下：

```
while (GetMessage (&msg, NULL, 0, 0))
     {
     if (!IsDialogMessage (hDlgModeless, &msg))
          {
          TranslateMessage (&msg) ;
          DispatchMessage  (&msg) ;
          }
     }
```

由于对话框是在程序进入消息循环前创建的并且直到程序结束之时才会被清除，所以 hDlgModeless 的值将总是有效的。我加入了如下的代码，以便你可能会往对话框的窗口过程中加入一段清除对话框的代码：

```
case WM_CLOSE:
    DestroyWindow (hDlg) ;
    hDlgModeless = 0;
    break;
```

在原来的 COLORS1 程序中，SetWindowText 在使用 itoa 将三个数值标签转换为文本之后才设置它们的值。语句为：

```
SetWindowText (hwndValue[i], itoa (color[i], szBuffer, 10)) ;
```

i 的值为当前处理的滚动条的 ID 号，hwndValue 是一个数组，它包含颜色数值的三个静态文本子窗口的窗口句柄。

新版本使用 SetDlgItemInt 为每个子窗口的每个文本字段设置一个号码：

```
SetDlgItemInt (hDlg, iCtrlID + 3, iColor[iIndex], FALSE) ;
```

（尽管 SetDlgItemInt 和与其对应的 GetDlgItemInt 在编辑控制中用得最多，它们也可以用来设置其它控制的文本字段，如静态文本控制等。）iCtrlID 变量是滚动条的 ID 号；给 ID 号加 3 使之变成对应的数字标签的 ID。第三个参数是颜色值。通常，第四个参数会被设置为 TRUE，以指出大于 32767 的数字将显示为负数，但是，对于这个程序，值的范围是从 0 到 256，所以这个参数没有意义。

在将 COLORS1 转换为 COLORS2 的过程中，我们把越来越多的工作交给了 Windows。旧版本调用了 CreateWindow10 次；而新版本只调用了 CreateWindow 和 CreateDialog 各一次。但是，如果你认为我们已经把调用 CreateWindow 的次数降到了最小，那么你就错了，请看下一个程序。

11.3.3 HEXCALC：窗口还是对话框

HEXCALC程序可能是偷懒的编程的典范之作，如图11-9中所示。这个程序完全不调用CreateWindow，也不处理WM_PAINT消息，不获取设备环境，也不处理鼠标消息。但是它只用了150行不到的源代码就构成了一个具有完整键盘和鼠标界面以及10种运算的十六进制计算器。计算器如图11-1所示。

HEXCALC.MAK

```
#-----------------------
# HEXCALC.MAK make file
#-----------------------

hexcalc.exe : hexcalc.obj hexcalc.res
     $(LINKER) $(GUIFLAGS) -OUT:hexcalc.exe hexcalc.obj hexcalc.res $(GUILIBS)

hexcalc.obj : hexcalc.c
     $(CC) $(CFLAGS) hexcalc.c

hexcalc.res : hexcalc.rc hexcalc.ico
     $(RC) $(RCVARS) hexcalc.rc
```

HEXCALC.C

```
/*----------------------------------------
   HEXCALC.C - Hexadecimal Calculator
               (c) Charles Petzold, 1996
  ----------------------------------------*/

#include <windows.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "HexCalc" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = DLGWINDOWEXTRA ;
```

```
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateDialog (hInstance, szAppName, 0, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void ShowNumber (HWND hwnd, UINT iNumber)
     {
     char szBuffer[20] ;

     SetDlgItemText (hwnd, VK_ESCAPE, strupr (ltoa (iNumber, szBuffer, 16))) ;
     }

DWORD CalcIt (UINT iFirstNum, int iOperation, UINT iNum)
     {
     switch (iOperation)
          {
          case '=' : return iNum ;
          case '+' : return iFirstNum +  iNum ;
          case '-' : return iFirstNum -  iNum ;
          case '*' : return iFirstNum *  iNum ;
          case '&' : return iFirstNum &  iNum ;
          case '|' : return iFirstNum |  iNum ;
          case '^' : return iFirstNum ^  iNum ;
          case '<' : return iFirstNum << iNum ;
          case '>' : return iFirstNum >> iNum ;
          case '/' : return iNum ? iFirstNum / iNum : UINT_MAX ;
          case '%' : return iNum ? iFirstNum % iNum : UINT_MAX ;
          default  : return 0 ;
          }
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL  bNewNumber = TRUE ;
     static int   iOperation = '=' ;
     static UINT  iNumber, iFirstNum ;
```

```
     HWND          hButton ;

     switch (iMsg)
          {
          case WM_KEYDOWN :                    // left arrow -> backspace
               if (wParam != VK_LEFT)
                    break ;
               wParam = VK_BACK ;
                                              // fall through
          case WM_CHAR :
               if ((wParam = toupper (wParam)) == VK_RETURN)
                    wParam = '=' ;

               hButton = GetDlgItem (hwnd, wParam) ;

               if (hButton != NULL)
                    {
                    SendMessage (hButton, BM_SETSTATE, 1, 0) ;
                    SendMessage (hButton, BM_SETSTATE, 0, 0) ;
                    }
               else
                    {
                    MessageBeep (0) ;
                    break ;
                    }
                                              // fall through
          case WM_COMMAND :
               SetFocus (hwnd) ;

               if (LOWORD (wParam) == VK_BACK)         // backspace
                    ShowNumber (hwnd, iNumber /= 16) ;

               else if (LOWORD (wParam) == VK_ESCAPE)  // escape
                    ShowNumber (hwnd, iNumber = 0) ;

               else if (isxdigit (LOWORD (wParam)))    // hex digit
                    {
                    if (bNewNumber)
                         {
                         iFirstNum = iNumber ;
                         iNumber = 0 ;
                         }
                    bNewNumber = FALSE ;

                    if (iNumber <= UINT_MAX >> 4)
                         ShowNumber (hwnd, iNumber = 16 * iNumber + wParam -
                              (isdigit (wParam) ? '0' : 'A' - 10)) ;
                    else
                         MessageBeep (0) ;
                    }
               else                                    // operation
                    {
                    if (!bNewNumber)
```

```
                    ShowNumber (hwnd, iNumber =
                        CalcIt (iFirstNum, iOperation, iNumber)) ;
                bNewNumber = TRUE ;
                iOperation = LOWORD (wParam) ;
                }
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

HEXCALC.RC

```
/*----------------------------
   HEXCALC.RC resource script
  ----------------------------*/

#include <windows.h>

HexCalc ICON hexcalc.ico

HexCalc DIALOG 32768, 0, 102, 122
     STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
     CLASS "HexCalc"
     CAPTION "Hex Calculator"
     {
     PUSHBUTTON "D",       68,  8,  24, 14, 14
     PUSHBUTTON "A",       65,  8,  40, 14, 14
     PUSHBUTTON "7",       55,  8,  56, 14, 14
     PUSHBUTTON "4",       52,  8,  72, 14, 14
     PUSHBUTTON "1",       49,  8,  88, 14, 14
     PUSHBUTTON "0",       48,  8, 104, 14, 14
     PUSHBUTTON "0",       27, 26,   4, 50, 14
     PUSHBUTTON "E",       69, 26,  24, 14, 14
     PUSHBUTTON "B",       66, 26,  40, 14, 14
     PUSHBUTTON "8",       56, 26,  56, 14, 14
     PUSHBUTTON "5",       53, 26,  72, 14, 14
     PUSHBUTTON "2",       50, 26,  88, 14, 14
     PUSHBUTTON "Back",     8, 26, 104, 32, 14
     PUSHBUTTON "C",       67, 44,  40, 14, 14
     PUSHBUTTON "F",       70, 44,  24, 14, 14
     PUSHBUTTON "9",       57, 44,  56, 14, 14
     PUSHBUTTON "6",       54, 44,  72, 14, 14
     PUSHBUTTON "3",       51, 44,  88, 14, 14
     PUSHBUTTON "+",       43, 62,  24, 14, 14
     PUSHBUTTON "-",       45, 62,  40, 14, 14
     PUSHBUTTON "*",       42, 62,  56, 14, 14
     PUSHBUTTON "/",       47, 62,  72, 14, 14
     PUSHBUTTON "%",       37, 62,  88, 14, 14
     PUSHBUTTON "Equals",  61, 62, 104, 32, 14
     PUSHBUTTON "&&",      38, 80,  24, 14, 14
```

```
PUSHBUTTON "|",       124, 80,  40, 14, 14
PUSHBUTTON "^",        94, 80,  56, 14, 14
PUSHBUTTON "<",        60, 80,  72, 14, 14
PUSHBUTTON ">",        62, 80,  88, 14, 14
}
```

HEXCALC.ICO

图 11-9 HEXCALC程序

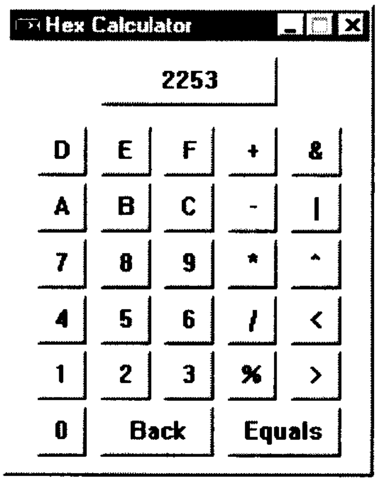


图 11-10 HEXCALC的显示

HEXCALC 是一个通常的中缀表达式计算器，使用 C 语言记法进行计算。它对无符号 32 位整数作加、减、乘、除和取余数，按位与、或和异或运算，还有左右位移运算。被 0 除将导致结果被设置为 FFFFFFFF 。

在 HEXCALC 中既可以使用鼠标又可以使用键盘。你从“点入”或者输入第一个数（最多 8 位十六进制数字）开始，然后输入运算符，然后是第二个数。接着，你可以通过单击 Equals 按钮或者按下等号键或回车键便可以显示运算结果。为了更正输入，你可以使用 Back 按钮、 Backspace 或者左箭头键。单击“显示”框或者按下 Esc 键即可清除当前的输入。

HEXCALC 比较奇怪的一点是，屏幕上显示的窗口似乎是通常的重叠式窗口与非模态对话框的混合体。一方面， HEXCALC 的所有消息都在函数的 WndProc 中处理，这个函数与通常的窗口过程相似，该函数返回一个长整数，它处理 WM_DESTROY 消息，调用

DefWindowProc ，就像普通的窗口过程一样。另一方面，窗口是在 WinMain 中调用 CreateDialog 并使用 HEXCALC.RC 中的对话框模板创建的。那么，HEXCALC 到底是一个普通的重叠式窗口呢，还是一个非模态对话框呢？

简单的回答是，对话框就是窗口。通常，窗口使用它自己内部的窗口过程处理对话框弹出式窗口的消息，然后，Windows 将这些消息传送给创建对话框的程序内的对话框过程。在 HEXCALC 中，我们让 Windows 使用对话框模板创建一个弹出式窗口，但是自己编程处理这个窗口的消息。

详细分析 HEXCALC.RC 能够揭示出这是如何做到的。对话框模板如下开始：

```
HexCalc DIALOG 32768, 0, 102, 122
    STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
    CLASS "HexCalc"
    CAPTION "Hex Calculator"
```

注意诸如 WS_OVERLAPPED 和 WS_MINIMIZEBOX 等标识符，我们可以将它们用在 CreateWindow 调用中以创建普通的窗口。CLASS 语句是这个对话框与曾经创建过的对话框之间最重要的区别。当前面的对话框模板省略了这个语句时，Windows 为对话框注册一个窗口类，并使用它自己的窗口过程处理对话框消息。这里，包含 CLASS 语句就告诉 Windows 将消息发送到其它地方——具体地说，就是发送到在"HexCalc"窗口类中指定的窗口过程。

"HexCalc"窗口类是在 HEXCALC 的 WinMain 函数中注册的，就象普通窗口的窗口类一样。但是，请注意这个十分重要的区别：WNDCLASS 结构的 cbWndExtra 字段设置为 DLGWINDOWEXTRA 。对于你自己注册的对话框过程，这是必需的。

在注册窗口类之后，WinMain 调用 CreateDialog：

```
hwnd = CreateDialog (hInstance, szAppName, 0, NULL) ;
```

第二个参数（串"HexCalc"）是对话框模板的名字。第三个参数通常是父窗口的窗口句柄，这里设置为 0，因为窗口没有父窗口。最后一个参数，通常是对话框过程的地址，这里不需要，因为 Windows 不会处理这些消息，因而也不会将消息发送给对话框过程。

这个 CreateDialog 调用与对话框模板一起，被 Windows 有效地转换为一个 CreateWindow 调用。该 CreateWindow 调用的功能与下面的调用等价：

```
hwnd = CreateWindow ("HexCalc", "Hex Calculaator",
    WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX,
    CW_USEDEFAULT, CW_USEDEFAULT,
    102 * 4 / cxChar, 122 * 8 / cyChar,
    NULL, NULL, hInstance, NULL) ;
```

其中，cxChar 和 cyChar 变量分别是系统字体字符的宽度和高度。

我们通过让 Windows 来进行 CreateWindow 调用而收获甚丰：Windows 不会在创建弹出式窗口 1 后就停止，它还会为对话框模板中定义的其它 29 个子窗口压入按钮控制调用 CreateWindow 。所有这些控制都给父窗口的窗口过程发送 WM_COMMAND 消息，该过程

正是 WndProc 。这是创建一个包含许多子窗口的窗口的很好的技巧。

11.3.4 创造性地使用控制 ID

HEXCALC 没有包含说明对话框模板中所有子窗口控制的 ID 号标识符的头文件。我们之所以可以不要这个文件，是因为每个压入按钮控制的 ID 号设置为出现在控制上的文本的 ASCII 码，这意味着， WndProc 可以完全相同地对待 WM_COMMAND 消息和 WM_CHAR 消息。在每种情况下， wParam 都是按钮的 ASCII 码。

当然，对键盘消息进行一些处理是必要的。 WndProc 捕获 WM_KEYDOWN 消息，将左箭头键转换为 Backspace 键。在处理 WM_CHAR 消息时， WndProc 将字符代码转换为大写， Enter 键转换为等号键的 ASCII 码。

WM_CHAR 消息的有效性是通过调用 GetDlgItem 来检验的。如果 GetDlgItem 函数返回 0 ，那么键盘字符不是对话框模板中定义的 ID 号之一。如果字符是 ID 之一，则通过给相应的按钮发送一对 BM_SETSTATE 消息，来使之闪烁：

```
hButton = (SetDlgItem (hwnd, wParam)) ;
if (hButton != NULL)
     {
     SendMessage (hButton, BM_SETSTATE, 1, 0) ;
     SendMessage (hButton, BM_SETSTATE, 0, 0) ;
     }
```

这样做，用最小的代价，给 HEXCALC 的键盘界面增色不少。

当 WndProc 处理 WM_COMMAND 消息时，它总是将输入焦点设置为父窗口：

```
case WM_COMMAND:
     SetFocus (hwnd) ;
```

否则，一旦使用鼠标单击某按钮，输入焦点就会切换到该按钮上。

11.4 通用对话框

Windows 的一个主要目的是推进标准的用户界面。对许多常用的菜单项来说，这发生得很快，几乎所有软件生产商都采用 Alt-File-Open 选择来打开一个文件。然而，实际的文件打开对话框却经常很不一样。

从 Windows 3.1 开始，对这个问题有了一个可用的解决方案，并且该方案在 Windows 95 中仍然支持，这是一种叫做“通用对话框库”的增强。这个库由几个函数组成，这些函数激活标准对话框来进行打开和保存文件、搜索和替换、选择颜色、选择字体（我都将在本章中进行讨论）以及打印（我将在第 15 章进行讨论。）等操作。

为了使用这些函数，你基本上都要初始化某一结构的各个字段，并将该结构的指针传送给通用对话框库的某个函数，该函数创建并显示对话框。当用户关闭对话框时，被调用

的函数将控制返回给程序，你可以从传送给它的结构中获得信息。

COMMDLG.H 头文件定义了使用这些对话框所需要的函数和结构。COMDLG32.LIB 是通用对话框输入库，COMDLG32.DLL 是一个动态链接库，它包含对话框模板和过程。

11.4.1 增强 POPPAD

当我们在第 10 章中往 POPPAD 中增加一个菜单时，留下了几个标准菜单选项没有实现。现在我们已经准备好在 POPPAD 中加入打开文件、读入文件以及在磁盘上保存被编辑的文件的功能。在处理中，我们还将在 POPPAD 中加入字体选择和搜索替换功能。

实现 POPPAD3 程序的文件如图 11-11 所示。

POPPAD3.MAK

```
#-----------------------
# POPPAD3.MAK make file
#-----------------------

poppad3.exe : poppad.obj  popfile.obj  popfind.obj \
              popfont.obj popprnt0.obj poppad.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad3.exe poppad.obj popfile.obj \
              popfind.obj popfont.obj popprnt0.obj poppad.res $(GUILIBS)

poppad.obj : poppad.c poppad.h
     $(CC) $(CFLAGS) poppad.c

popfile.obj : popfile.c
     $(CC) $(CFLAGS) popfile.c

popfind.obj : popfind.c
     $(CC) $(CFLAGS) popfind.c

popfont.obj : popfont.c
     $(CC) $(CFLAGS) popfont.c

popprnt0.obj : popprnt0.c
     $(CC) $(CFLAGS) popprnt0.c

poppad.res : poppad.rc poppad.h poppad.ico
     $(RC) $(RCVARS) poppad.rc
```

POPPAD.C

```
/*---------------------------------------
   POPPAD.C - Popup Editor
              (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <stdlib.h>
#include "poppad.h"
```

```
#define EDITID   1
#define UNTITLED "(untitled)"

LRESULT CALLBACK WndProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK AboutDlgProc (HWND, UINT, WPARAM, LPARAM) ;

          // Functions in POPFILE.C

void PopFileInitialize (HWND) ;
BOOL PopFileOpenDlg    (HWND, PSTR, PSTR) ;
BOOL PopFileSaveDlg    (HWND, PSTR, PSTR) ;
BOOL PopFileRead       (HWND, PSTR) ;
BOOL PopFileWrite      (HWND, PSTR) ;

          // Functions in POPFIND.C

HWND PopFindFindDlg     (HWND) ;
HWND PopFindReplaceDlg  (HWND) ;
BOOL PopFindFindText    (HWND, int *, LPFINDREPLACE) ;
BOOL PopFindReplaceText (HWND, int *, LPFINDREPLACE) ;
BOOL PopFindNextText    (HWND, int *) ;
BOOL PopFindValidFind   (void) ;

          // Functions in POPFONT.C

void PopFontInitialize   (HWND) ;
BOOL PopFontChooseFont   (HWND) ;
void PopFontSetFont      (HWND) ;
void PopFontDeinitialize (void) ;

          // Functions in POPPRNT.C

BOOL PopPrntPrintFile (HINSTANCE, HWND, HWND, PSTR) ;

          // Global variables

static char szAppName[] = "PopPad" ;
static HWND hDlgModeless ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     MSG         msg ;
     HWND        hwnd ;
     HACCEL      hAccel ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
```

```
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, NULL,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, szCmdLine) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     hAccel = LoadAccelerators (hInstance, szAppName) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          if (hDlgModeless == NULL || !IsDialogMessage (hDlgModeless, &msg))
               {
               if (!TranslateAccelerator (hwnd, hAccel, &msg))
                    {
                    TranslateMessage (&msg) ;
                    DispatchMessage (&msg) ;
                    }
               }
          }
     return msg.wParam ;
     }

void DoCaption (HWND hwnd, char *szTitleName)
     {
     char szCaption[64 + _MAX_FNAME + _MAX_EXT] ;

     wsprintf (szCaption, "%s - %s", szAppName,
               szTitleName[0] ? szTitleName : UNTITLED) ;

     SetWindowText (hwnd, szCaption) ;
     }

void OkMessage (HWND hwnd, char *szMessage, char *szTitleName)
     {
     char szBuffer[64 + _MAX_FNAME + _MAX_EXT] ;

     wsprintf (szBuffer, szMessage, szTitleName[0] ? szTitleName : UNTITLED) ;

     MessageBox (hwnd, szBuffer, szAppName, MB_OK | MB_ICONEXCLAMATION) ;
     }

short AskAboutSave (HWND hwnd, char *szTitleName)
     {
     char szBuffer[64 + _MAX_FNAME + _MAX_EXT] ;
     int  iReturn ;

     wsprintf (szBuffer, "Save current changes in %s?",
               szTitleName[0] ? szTitleName : UNTITLED) ;
```

```
     iReturn = MessageBox (hwnd, szBuffer, szAppName,
                           MB_YESNOCANCEL | MB_ICONQUESTION) ;

     if (iReturn == IDYES)
          if (!SendMessage (hwnd, WM_COMMAND, IDM_SAVE, 0L))
               iReturn = IDCANCEL ;

     return iReturn ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL      bNeedSave = FALSE ;
     static char      szFileName[_MAX_PATH] ;
     static char      szTitleName[_MAX_FNAME + _MAX_EXT] ;
     static HINSTANCE hInst ;
     static HWND      hwndEdit ;
     static int       iOffset ;
     static UINT      iMsgFindReplace ;
     int              iSelBeg, iSelEnd, iEnable ;
     LPFINDREPLACE    pfr ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInst = ((LPCREATESTRUCT) lParam) -> hInstance ;

                         // Create the edit control child window

               hwndEdit = CreateWindow ("edit", NULL,
                         WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL |
                              WS_BORDER | ES_LEFT | ES_MULTILINE |
                              ES_NOHIDESEL | ES_AUTOHSCROLL | ES_AUTOVSCROLL,
                         0, 0, 0, 0,
                         hwnd, (HMENU) EDITID, hInst, NULL) ;

               SendMessage (hwndEdit, EM_LIMITTEXT, 32000, 0L) ;

                         // Initialize common dialog box stuff

               PopFileInitialize (hwnd) ;
               PopFontInitialize (hwndEdit) ;

               iMsgFindReplace = RegisterWindowMessage (FINDMSGSTRING) ;

                         // Process command line

               lstrcpy (szFileName, (PSTR)
                        (((LPCREATESTRUCT) lParam)->lpCreateParams)) ;

               if (strlen (szFileName) > 0)
                    {
                    GetFileTitle (szFileName, szTitleName,
                                  sizeof (szTitleName)) ;

                    if (!PopFileRead (hwndEdit, szFileName))
                         OkMessage (hwnd, "File %s cannot be read!",
```

```
                                  szTitleName) ;
               }

          DoCaption (hwnd, szTitleName) ;
          return 0 ;

     case WM_SETFOCUS :
          SetFocus (hwndEdit) ;
          return 0 ;

     case WM_SIZE :
          MoveWindow (hwndEdit, 0, 0, LOWORD (lParam),
                                      HIWORD (lParam), TRUE) ;
          return 0 ;

     case WM_INITMENUPOPUP :
          switch (lParam)
               {
               case 1 :          // Edit menu

                         // Enable Undo if edit control can do it

                    EnableMenuItem ((HMENU) wParam, IDM_UNDO,
                         SendMessage (hwndEdit, EM_CANUNDO, 0, 0L) ?
                              MF_ENABLED : MF_GRAYED) ;

                         // Enable Paste if text is in the clipboard

                    EnableMenuItem ((HMENU) wParam, IDM_PASTE,
                         IsClipboardFormatAvailable (CF_TEXT) ?
                              MF_ENABLED : MF_GRAYED) ;

                         // Enable Cut, Copy, and Del if text is selected

                    SendMessage (hwndEdit, EM_GETSEL, (WPARAM) &iSelBeg,
                                                      (LPARAM) &iSelEnd) ;

                    iEnable = iSelBeg != iSelEnd ? MF_ENABLED : MF_GRAYED ;

                    EnableMenuItem ((HMENU) wParam, IDM_CUT,   iEnable) ;
                    EnableMenuItem ((HMENU) wParam, IDM_COPY,  iEnable) ;
                    EnableMenuItem ((HMENU) wParam, IDM_CLEAR, iEnable) ;
                    break ;

               case 2 :          // Search menu

                         // Enable Find, Next, and Replace if modeless
                         //   dialogs are not already active

                    iEnable = hDlgModeless == NULL ?
                                   MF_ENABLED : MF_GRAYED ;

                    EnableMenuItem ((HMENU) wParam, IDM_FIND,    iEnable) ;
                    EnableMenuItem ((HMENU) wParam, IDM_NEXT,    iEnable) ;
                    EnableMenuItem ((HMENU) wParam, IDM_REPLACE, iEnable) ;
                    break ;
               }
```

```
          return 0 ;

     case WM_COMMAND :
                         // Messages from edit control

          if (lParam && LOWORD (wParam) == EDITID)
               {
               switch (HIWORD (wParam))
                    {
                    case EN_UPDATE :
                         bNeedSave = TRUE ;
                         return 0 ;

                    case EN_ERRSPACE :
                    case EN_MAXTEXT :
                         MessageBox (hwnd, "Edit control out of space.",
                                   szAppName, MB_OK | MB_ICONSTOP) ;
                         return 0 ;
                    }
               break ;
               }

          switch (LOWORD (wParam))
               {
                         // Messages from File menu

               case IDM_NEW :
                    if (bNeedSave && IDCANCEL ==
                              AskAboutSave (hwnd, szTitleName))
                         return 0 ;

                    SetWindowText (hwndEdit, "\0") ;
                    szFileName[0]  = '\0' ;
                    szTitleName[0] = '\0' ;
                    DoCaption (hwnd, szTitleName) ;
                    bNeedSave = FALSE ;
                    return 0 ;

               case IDM_OPEN :
                    if (bNeedSave && IDCANCEL ==
                              AskAboutSave (hwnd, szTitleName))
                         return 0 ;

                    if (PopFileOpenDlg (hwnd, szFileName, szTitleName))
                         {
                         if (!PopFileRead (hwndEdit, szFileName))
                              {
                              OkMessage (hwnd, "Could not read file %s!",
                                                  szTitleName) ;
                              szFileName[0]  = '\0' ;
                              szTitleName[0] = '\0';
                              }
                         }

                    DoCaption (hwnd, szTitleName) ;
                    bNeedSave = FALSE ;
                    return 0 ;
```

```
case IDM_SAVE :
     if (szFileName[0])
          {
          if (PopFileWrite (hwndEdit, szFileName))
               {
               bNeedSave = FALSE ;
               return 1 ;
               }
          else
               OkMessage (hwnd, "Could not write file %s",
                                 szTitleName) ;
          return 0 ;
          }
                                   // fall through
case IDM_SAVEAS :
     if (PopFileSaveDlg (hwnd, szFileName, szTitleName))
          {
          DoCaption (hwnd, szTitleName) ;

          if (PopFileWrite (hwndEdit, szFileName))
               {
               bNeedSave = FALSE ;
               return 1 ;
               }
          else
               OkMessage (hwnd, "Could not write file %s",
                                 szTitleName) ;
          }
     return 0 ;

case IDM_PRINT :
     if (!PopPrntPrintFile (hInst, hwnd, hwndEdit,
                              szTitleName))
          OkMessage (hwnd, "Could not print file %s",
                    szTitleName) ;
     return 0 ;

case IDM_EXIT :
     SendMessage (hwnd, WM_CLOSE, 0, 0) ;
     return 0 ;

          // Messages from Edit menu

case IDM_UNDO :
     SendMessage (hwndEdit, WM_UNDO, 0, 0) ;
     return 0 ;

case IDM_CUT :
     SendMessage (hwndEdit, WM_CUT, 0, 0) ;
     return 0 ;

case IDM_COPY :
     SendMessage (hwndEdit, WM_COPY, 0, 0) ;
     return 0 ;

case IDM_PASTE :
```

```
                    SendMessage (hwndEdit, WM_PASTE, 0, 0) ;
                    return 0 ;

               case IDM_CLEAR :
                    SendMessage (hwndEdit, WM_CLEAR, 0, 0) ;
                    return 0 ;

               case IDM_SELALL :
                    SendMessage (hwndEdit, EM_SETSEL, 0, -1) ;
                    return 0 ;

                         // Messages from Search menu

               case IDM_FIND :
                    SendMessage (hwndEdit, EM_GETSEL, NULL,
                                               (LPARAM) &iOffset) ;

                    hDlgModeless = PopFindFindDlg (hwnd) ;
                    return 0 ;

               case IDM_NEXT :
                    SendMessage (hwndEdit, EM_GETSEL, NULL,
                                               (LPARAM) &iOffset) ;

                    if (PopFindValidFind ())
                         PopFindNextText (hwndEdit, &iOffset) ;
                    else
                         hDlgModeless = PopFindFindDlg (hwnd) ;
                    return 0 ;

               case IDM_REPLACE :
                    SendMessage (hwndEdit, EM_GETSEL, NULL,
                                               (LPARAM) &iOffset) ;

                    hDlgModeless = PopFindReplaceDlg (hwnd) ;
                    return 0 ;

               case IDM_FONT :
                    if (PopFontChooseFont (hwnd))
                         PopFontSetFont (hwndEdit) ;

                    return 0 ;

                         // Messages from Help menu

               case IDM_HELP :
                    OkMessage (hwnd, "Help not yet implemented!", "\0") ;
                    return 0 ;

               case IDM_ABOUT :
                    DialogBox (hInst, "AboutBox", hwnd, AboutDlgProc) ;
                    return 0 ;
               }
          break ;

     case WM_CLOSE :
          if (!bNeedSave || IDCANCEL != AskAboutSave (hwnd, szTitleName))
```

```
                DestroyWindow (hwnd) ;

            return 0 ;

        case WM_QUERYENDSESSION :
            if (!bNeedSave || IDCANCEL != AskAboutSave (hwnd, szTitleName))
                return 1 ;

            return 0 ;

        case WM_DESTROY :
            PopFontDeinitialize () ;
            PostQuitMessage (0) ;
            return 0 ;

        default :
                    // Process "Find-Replace" iMsgs

            if (iMsg == iMsgFindReplace)
                {
                pfr = (LPFINDREPLACE) lParam ;

                if (pfr->Flags & FR_DIALOGTERM)
                    hDlgModeless = NULL ;

                if (pfr->Flags & FR_FINDNEXT)
                    if (!PopFindFindText (hwndEdit, &iOffset, pfr))
                        OkMessage (hwnd, "Text not found!", "\0") ;

                if (pfr->Flags & FR_REPLACE ||
                    pfr->Flags & FR_REPLACEALL)
                    if (!PopFindReplaceText (hwndEdit, &iOffset, pfr))
                        OkMessage (hwnd, "Text not found!", "\0") ;

                if (pfr->Flags & FR_REPLACEALL)
                    while (PopFindReplaceText (hwndEdit, &iOffset, pfr)) ;

                return 0 ;
                }
            break ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }

BOOL CALLBACK AboutDlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
    switch (iMsg)
        {
        case WM_INITDIALOG :
            return TRUE ;

        case WM_COMMAND :
            switch (LOWORD (wParam))
                {
                case IDOK :
                    EndDialog (hDlg, 0) ;
                    return TRUE ;
```

```
                    }
               break ;
          }
     return FALSE ;
     }
```

POPFILE.C

```
/*-----------------------------------------
   POPFILE.C - Popup Editor File Functions
  -----------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <stdlib.h>
#include <stdio.h>

static OPENFILENAME ofn ;

void PopFileInitialize (HWND hwnd)
     {
     static char szFilter[] = "Text Files (*.TXT)\0*.txt\0"  \
                              "ASCII Files (*.ASC)\0*.asc\0" \
                              "All Files (*.*)\0*.*\0\0";

     ofn.lStructSize       = sizeof (OPENFILENAME) ;
     ofn.hwndOwner         = hwnd ;
     ofn.hInstance         = NULL ;
     ofn.lpstrFilter       = szFilter ;
     ofn.lpstrCustomFilter = NULL ;
     ofn.nMaxCustFilter    = 0 ;
     ofn.nFilterIndex      = 0 ;
     ofn.lpstrFile         = NULL ;          // Set in Open and Close functions
     ofn.nMaxFile          = _MAX_PATH ;
     ofn.lpstrFileTitle    = NULL ;          // Set in Open and Close functions
     ofn.nMaxFileTitle     = _MAX_FNAME + _MAX_EXT ;
     ofn.lpstrInitialDir   = NULL ;
     ofn.lpstrTitle        = NULL ;
     ofn.Flags             = 0 ;             // Set in Open and Close functions
     ofn.nFileOffset       = 0 ;
     ofn.nFileExtension    = 0 ;
     ofn.lpstrDefExt       = "txt" ;
     ofn.lCustData         = 0L ;
     ofn.lpfnHook          = NULL ;
     ofn.lpTemplateName    = NULL ;
     }

BOOL PopFileOpenDlg (HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
     {
     ofn.hwndOwner         = hwnd ;
     ofn.lpstrFile         = pstrFileName ;
     ofn.lpstrFileTitle    = pstrTitleName ;
     ofn.Flags             = OFN_HIDEREADONLY | OFN_CREATEPROMPT ;

     return GetOpenFileName (&ofn) ;
     }
```

```
BOOL PopFileSaveDlg (HWND hwnd, PSTR pstrFileName, PSTR pstrTitleName)
     {
     ofn.hwndOwner          = hwnd ;
     ofn.lpstrFile          = pstrFileName ;
     ofn.lpstrFileTitle     = pstrTitleName ;
     ofn.Flags              = OFN_OVERWRITEPROMPT ;

     return GetSaveFileName (&ofn) ;
     }

static long PopFileLength (FILE *file)
     {
     int iCurrentPos, iFileLength ;

     iCurrentPos = ftell (file) ;

     fseek (file, 0, SEEK_END) ;

     iFileLength = ftell (file) ;

     fseek (file, iCurrentPos, SEEK_SET) ;

     return iFileLength ;
     }

BOOL PopFileRead (HWND hwndEdit, PSTR pstrFileName)
     {
     FILE *file ;
     int   iLength ;
     PSTR  pstrBuffer ;

     if (NULL == (file = fopen (pstrFileName, "rb")))
          return FALSE ;

     iLength = PopFileLength (file) ;

     if (NULL == (pstrBuffer = (PSTR) malloc (iLength)))
          {
          fclose (file) ;
          return FALSE ;
          }

     fread (pstrBuffer, 1, iLength, file) ;
     fclose (file) ;
     pstrBuffer[iLength] = '\0' ;

     SetWindowText (hwndEdit, pstrBuffer) ;
     free (pstrBuffer) ;

     return TRUE ;
     }

BOOL PopFileWrite (HWND hwndEdit, PSTR pstrFileName)
     {
     FILE *file ;
     int   iLength ;
     PSTR  pstrBuffer ;
```

```
     if (NULL == (file = fopen (pstrFileName, "wb")))
          return FALSE ;

     iLength = GetWindowTextLength (hwndEdit) ;

     if (NULL == (pstrBuffer = (PSTR) malloc (iLength + 1)))
          {
          fclose (file) ;
          return FALSE ;
          }

     GetWindowText (hwndEdit, pstrBuffer, iLength +1) ;

     if (iLength != (int) fwrite (pstrBuffer, 1, iLength, file))
          {
          fclose (file) ;
          free (pstrBuffer) ;
          return FALSE ;
          }

     fclose (file) ;
     free (pstrBuffer) ;

     return TRUE ;
     }
```

POPFIND.C

```
/*-----------------------------------------------------
   POPFIND.C -- Popup Editor Search and Replace Functions
  -----------------------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <string.h>
#define MAX_STRING_LEN   256

static char szFindText [MAX_STRING_LEN] ;
static char szReplText [MAX_STRING_LEN] ;

HWND PopFindFindDlg (HWND hwnd)
     {
     static FINDREPLACE fr ;       // must be static for modeless dialog!!!

     fr.lStructSize      = sizeof (FINDREPLACE) ;
     fr.hwndOwner        = hwnd ;
     fr.hInstance        = NULL ;
     fr.Flags            = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD ;
     fr.lpstrFindWhat    = szFindText ;
     fr.lpstrReplaceWith = NULL ;
     fr.wFindWhatLen     = sizeof (szFindText) ;
     fr.wReplaceWithLen  = 0 ;
     fr.lCustData        = 0 ;
     fr.lpfnHook         = NULL ;
     fr.lpTemplateName   = NULL ;
```

```
     return FindText (&fr) ;
     }

HWND PopFindReplaceDlg (HWND hwnd)
     {
     static FINDREPLACE fr ;       // must be static for modeless dialog!!!

     fr.lStructSize      = sizeof (FINDREPLACE) ;
     fr.hwndOwner        = hwnd ;
     fr.hInstance        = NULL ;
     fr.Flags            = FR_HIDEUPDOWN | FR_HIDEMATCHCASE | FR_HIDEWHOLEWORD ;
     fr.lpstrFindWhat    = szFindText ;
     fr.lpstrReplaceWith = szReplText ;
     fr.wFindWhatLen     = sizeof (szFindText) ;
     fr.wReplaceWithLen  = sizeof (szReplText) ;
     fr.lCustData        = 0 ;
     fr.lpfnHook         = NULL ;
     fr.lpTemplateName   = NULL ;

     return ReplaceText (&fr) ;
     }

BOOL PopFindFindText (HWND hwndEdit, int *piSearchOffset, LPFINDREPLACE pfr)
     {
     int  iLength, iPos ;
     PSTR pstrDoc, pstrPos ;

               // Read in the edit document

     iLength = GetWindowTextLength (hwndEdit) ;

     if (NULL == (pstrDoc = (PSTR) malloc (iLength + 1)))
          return FALSE ;

     GetWindowText (hwndEdit, pstrDoc, iLength + 1) ;

               // Search the document for the find string

     pstrPos = strstr (pstrDoc + *piSearchOffset, pfr->lpstrFindWhat) ;
     free (pstrDoc) ;

               // Return an error code if the string cannot be found

     if (pstrPos == NULL)
          return FALSE ;

               // Find the position in the document and the new start offset

     iPos = pstrPos - pstrDoc ;
     *piSearchOffset = iPos + strlen (pfr->lpstrFindWhat) ;

               // Select the found text

     SendMessage (hwndEdit, EM_SETSEL, iPos, *piSearchOffset) ;
     SendMessage (hwndEdit, EM_SCROLLCARET 0, 0) ;

     return TRUE ;
```

```
     }

BOOL PopFindNextText (HWND hwndEdit, int *piSearchOffset)
     {
     FINDREPLACE fr ;

     fr.lpstrFindWhat = szFindText ;

     return PopFindFindText (hwndEdit, piSearchOffset, &fr) ;
     }

BOOL PopFindReplaceText (HWND hwndEdit, int *piSearchOffset, LPFINDREPLACE pfr)
     {
               // Find the text

     if (!PopFindFindText (hwndEdit, piSearchOffset, pfr))
          return FALSE ;

               // Replace it

     SendMessage (hwndEdit, EM_REPLACESEL, 0, (LPARAM) pfr->lpstrReplaceWith) ;

     return TRUE ;
     }

BOOL PopFindValidFiMnd (void)
     {
     return *szFindText != '\0' ;
     }
```

POPFONT.C

```
/*-----------------------------------------
   POPFONT.C - Popup Editor Font Functions
  -----------------------------------------*/

#include <windows.h>
#include <commdlg.h>

static LOGFONT logfont ;
static HFONT   hFont ;

BOOL PopFontChooseFont (HWND hwnd)
     {
     CHOOSEFONT cf ;

     cf.lStructSize      = sizeof (CHOOSEFONT) ;
     cf.hwndOwner        = hwnd ;
     cf.hDC              = NULL ;
     cf.lpLogFont        = &logfont ;
     cf.iPointSize       = 0 ;
     cf.Flags            = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS
                                                  | CF_EFFECTS ;
     cf.rgbColors        = 0L ;
     cf.lCustData        = 0L ;
     cf.lpfnHook         = NULL ;
     cf.lpTemplateName   = NULL ;
```

```
     cf.hInstance        = NULL ;
     cf.lpszStyle        = NULL ;
     cf.nFontType        = 0 ;              // Returned from ChooseFont
     cf.nSizeMin         = 0 ;
     cf.nSizeMax         = 0 ;

     return ChooseFont (&cf) ;
     }

void PopFontInitialize (HWND hwndEdit)
     {
     GetObject (GetStockObject (SYSTEM_FONT), sizeof (LOGFONT),
                                          (PSTR) &logfont) ;
     hFont = CreateFontIndirect (&logfont) ;
     SendMessage (hwndEdit, WM_SETFONT, (WPARAM) hFont, 0) ;
     }

void PopFontSetFont (HWND hwndEdit)
     {
     HFONT hFontNew ;
     /RECT rect ;

     hFontNew = CreateFontIndirect (&logfont) ;
     SendMessage (hwndEdit, WM_SETFONT, (WPARAM) hFontNew, 0) ;
     DeleteObject (hFont) ;
     hFont = hFontNew ;
     GetClientRect (hwndEdit, &rect) ;
     InvalidateRect (hwndEdit, &rect, TRUE);
     }

void PopFontDeinitialize (void)
     {
     DeleteObject (hFont) ;
     }
```

POPPRNT0.C

```
/*---------------------------------------------------------------
   POPPRNT0.C - Popup Editor Printing Functions (dummy version)
  ---------------------------------------------------------------*/

#include <windows.h>

BOOL PopPrntPrintFile (HINSTANCE hInst, HWND hwnd, HWND hwndEdit,
                                    PSTR pstrTitleName)
     {
     return FALSE ;
     }
```

POPPAD.RC

```
/*---------------------------
   POPPAD.RC resource script
  ---------------------------*/

#include <windows.h>
#include "poppad.h"
```

```
PopPad ICON "poppad.ico"

PopPad MENU
     {
     POPUP "&File"
          {
          MENUITEM "&New\tCtrl+N",          IDM_NEW
          MENUITEM "&Open...\tCtrl+O",      IDM_OPEN
          MENUITEM "&Save\tCtrl+S",         IDM_SAVE
          MENUITEM "Save &As...",           IDM_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print...\tCtrl+P",     IDM_PRINT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                 IDM_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",         IDM_UNDO
          MENUITEM SEPARATOR
          MENUITEM "Cu&t\tCtrl+X",          IDM_CUT
          MENUITEM "&Copy\tCtrl+C",         IDM_COPY
          MENUITEM "&Paste\tCtrl+V",        IDM_PASTE
          MENUITEM "De&lete\tDel",          IDM_CLEAR
          MENUITEM SEPARATOR
          MENUITEM "&Select All",           IDM_SELALL
          }
     POPUP "&Search"
          {
          MENUITEM "&Find...\tCtrl+F",      IDM_FIND
          MENUITEM "Find &Next\tF3",        IDM_NEXT
          MENUITEM "R&eplace...\tCtrl+R",   IDM_REPLACE
          }
     POPUP "F&ormat"
          {
          MENUITEM "&Font...",              IDM_FONT
          }
     POPUP "&Help"
          {
          MENUITEM "&Help",                 IDM_HELP
          MENUITEM "&About PopPad...",      IDM_ABOUT
          }
     }

PopPad ACCELERATORS
     {
     "^N",         IDM_NEW
     "^O",         IDM_OPEN
     "^S",         IDM_SAVE
     "^P",         IDM_PRINT
     "^Z",         IDM_UNDO
     VK_BACK,      IDM_UNDO,    VIRTKEY, ALT
     "^X",         IDM_CUT
     VK_DELETE,    IDM_CUT,     VIRTKEY, SHIFT
     "^C",         IDM_COPY
     VK_INSERT,    IDM_COPY,    VIRTKEY, CONTROL
     "^V",         IDM_PASTE
```

```
     VK_INSERT,    IDM_PASTE, VIRTKEY, SHIFT
     VK_DELETE,    IDM_CLEAR, VIRTKEY
     "^F",         IDM_FIND
     VK_F3,        IDM_NEXT,  VIRTKEY
     "^R",         IDM_REPLACE
     VK_F1,        IDM_HELP,  VIRTKEY
     }

AboutBox DIALOG  20, 20, 160, 80
     STYLE WS_POPUP | WS_DLGFRAME
     {
     CTEXT "PopPad"                              -1,   0, 12, 160,  8
     ICON  "PopPad"                              -1,   8,  8,   0,  0
     CTEXT "Popup Editor for Microsoft Windows"  -1,   0, 36, 160,  8
     CTEXT "Copyright (c) Charles Petzold, 1996" -1,   0, 48, 160,  8
     DEFPUSHBUTTON    K"                       IDOK,  64, 60,  32, 14, WS_GROUP
     }

PrintDlgBox DIALOG 20, 20, 100, 76
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
     CAPTION "PopPad"
     {
     CTEXT "Sending",                  -1,  0, 10, 100,  8
     CTEXT "",                  IDD_FNAME,  0, 20, 100,  8
     CTEXT "to print spooler.",         -1,  0, 30, 100,  8
     DEFPUSHBUTTON  "Cancel",   IDCANCEL, 34, 50,  32, 14, WS_GROUP
     }
```

POPPAD.H

```
/*---------------------
   POPPAD.H header file
  ---------------------*/

#define IDM_NEW        10
#define IDM_OPEN       11
#define IDM_SAVE       12
#define IDM_SAVEAS     13
#define IDM_PRINT      14
#define IDM_EXIT       15

#define IDM_UNDO       20
#define IDM_CUT        21
#define IDM_COPY       22
#define IDM_PASTE      23
#define IDM_CLEAR      24
#define IDM_SELALL     25

#define IDM_FIND       30
#define IDM_NEXT       31
#define IDM_REPLACE    32

#define IDM_FONT       40

#define IDM_HELP       50
#define IDM_ABOUT      51
```

```
#define IDD_FNAME        10
```

POPPAD.ICO

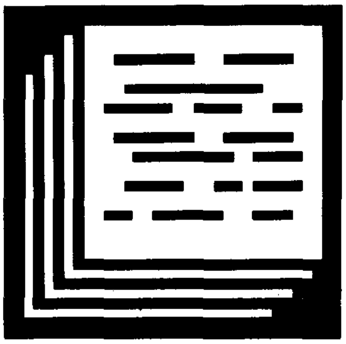


图 11-11 POPPAD3程序

为了避免在第 15 章中重复源代码，我在 POPPAD.RC 的菜单中加入了打印项目和一些其它的支持。

POPPAD.C 包含了程序中所有的基本源代码。POPFILE.C 具有激活 File Open 和 File Save 对话框的代码，它还包含文件 I/O 例程。POPFIND.C 中包含了查找和替换逻辑。POPFONT.C 和 POPPRNT0.C 不完成什么工作：在第 15 章中将使用 POPPRNT.C 替换 POPPRNT0.C 以创建最终的 POPPAD 程序。

让我们先来看一看 POPPAD.C 。你会注意到，在 WinMain 中， szCmdLine 参数用作 CreateWindow 调用的最后一个字段，这个串可能包含一个文件名，它是在程序执行时作为命令行参数输入的。在 WndProc 中处理 WM_CREATE 消息时，这个文件名被传送给位于 POPFILE.C 中的 PopFileRead 函数。

POPPAD.C 保存两个文件名串：第一个（以名字 szFileName 保存在 WndProc 中）是完整的驱动器、路径加上文件名。第二个（保存为 szTitleName ）是文件名本身，它用于 POPPAD3 的 Docaption 函数中，以便在窗口的标题条上显示文件名；也用在 OKMessage 函数和 AskAboutSave 函数中，以便显示消息框给用户。

POPFILE.C 包含了几个显示 File Open 和 File Save 对话框以及实际执行文件 I/O 的函数。对话框是使用函数 GetOpenFileName 和 GetSaveFileName 来显示的，这两个函数在通用对话框动态链接库（ COMDLG32.DLL ）中，都使用一个类型为 OPENFILENAME 的结构，这个结构在 COMMDLG.H 中定义。在 POPFILE.C 中，使用了一个该结构类型的全局变量，取名为 ofn 。 ofn 的大多数字段在 PopFileInitialize 函数中被初始化， POPPAD.C 在 WndProc 中处理 WM_CREATE 消息时调用该函数。

将 ofn 作为静态全局结构变量比较方便，因为 GetOpenFileName 和 GetSaveFileName 给该结构返回的一些信息将在以后调用这些函数时用到。

尽管通用对话框具有许多选项——包括设置自己的对话框模板以及为对话框过程增加“钩子(hook)”—— POPFILE.C 中使用的 File Open 和 File Save 对话框是最基本的。OPENFILENAME 结构中被设置的字段只有 lStructSize （结构的长度）、 hwndOwner （对话框拥有者）、 lpstrFilter （下面将简要讨论）、 lpstrFile 和 nMaxFile （指向接收完整文件名的缓冲区指针和该缓冲区的大小）、 lpstrFileTitle 和 nMaxFielTitle （文件名缓冲区及

其大小）、Flags（设置对话框的选项）和 lpstrDefExt（如果用户在对话框中输入文件名时不指定文件扩展名，那么它就是缺省的文件扩展名）。

当用户在 File 菜单中选择 Open 时，POPPAD2 调用 POPFILE 的 PopFileOpenDlg 函数，将窗口句柄、一个指向文件名缓冲区的指针和一个指向文件标题缓冲区的指针传给它。PopFileOpenDlg 恰当地设置 OPENFILENAME 结构的 wndOwner、lpstrFile 和 lpstrFileTitle 字段，将 Flags 设置为 OFN_HIDEREADONLY | OFN_CREATEPROMPT，然后调用 GetOpenFileName，显示如图 11-12 所示的普通对话框。缺省的文件打开对话框包括一个选择框，它允许用户指定文件以"只读方式"打开；OFN_HIDEREADONLY 标志导致 GetOpenFileName 不显示这个选择框。

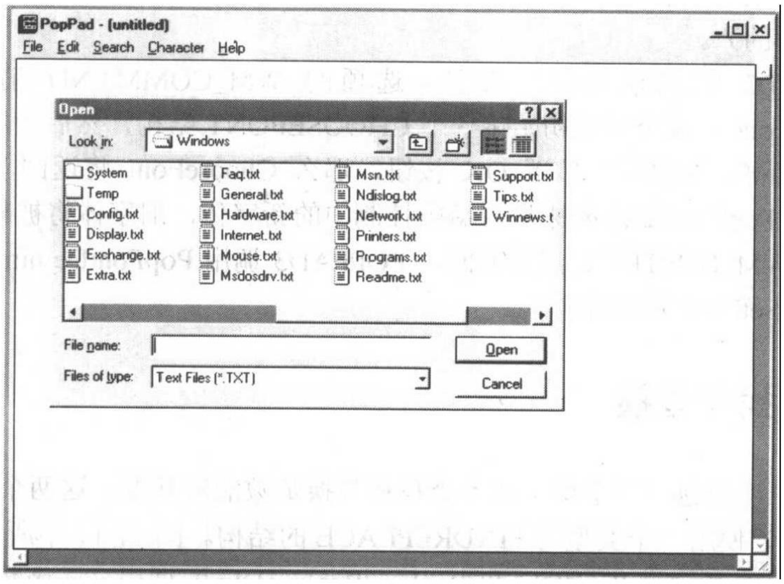


图 11-12 File Open对话框

当用户结束这个对话框时，GetOpenFileName 函数将返回。OFN_CREATEPROMPT 标志指示 GetOpenFileName 显示一个消息框，询问用户如果所选文件不存在，是否创建该文件。

左下角的组合框列出了将要显示在文件列表中的文件的类型，此列表框被称为"过滤器"，用户可以通过从组合框列表中选择另一种文件类型来改变过滤器。在 POPFILE.C 的 PopFileInitialize 函数中，我在变量 szFilter 中为三种类型的文件定义了一个过滤器：带有 TXT 扩展名的文本文件、带 ASC 扩展名的 ASCII 文件和所有文件。OPENFILENAME 结构的 lpstrFilter 字段被设置来指向这个过滤器。

如果用户在对话框处于活动状态时改变了过滤器，那么 OPENFILENAME 的 nFilterIndex 字段反映出用户的选择。由于该结构是静态变量，下次激活对话框时，过滤器将被设置为选中的文件类型。

POPFILE.C 中的 PopFileSaveDlg 函数与 PopFileOpenDlg 相似，它将 Flags 参数设置为 OFN_OVERWRITEPROMPT，并调用 GetSaveFileName 激活 File Save 对话框。OFN_OVERWRITEPROMPT 标志导致显示一个消息框，如果被选文件已经存在，那么将询问用户是否覆盖该文件。POPFILE.C 的其它函数使用标准 C 库函数执行文件 I/O（我将

在第 13 章中更详细地讨论文件 I/O ）。

11.4.2 改变字体

在第 4 章中，我曾经答应将向你说明你的程序如何可以简单地允许一个用户选择字体，这在 POPFONT.C 中显示。

在 WM_CREATE 消息处理期间， POPPAD3 调用 POPFONT.C 中的 PopFontInitialize 。这个函数获取一个基于系统字体的 LOGFONT 结构，由此创建一种字体，并向编辑控制发送一条 WM_SETFONT 消息来设置一种新的字体（缺省编辑控制字体是系统字体，而 PopFontInitialize 为编辑控制创建一种新的字体，因为最终该字体将被删除，而删除现有系统字体是不明智的）。

当 POPPAD3 收到来自程序的字体选项的 WM_COMMAND 消息时，它调用 PopFontChooseFont 。这个函数初始化一个 CHOOSEFONT 结构，然后调用 ChooseFont 显示字体选择对话框。如果用户按下 OK 按钮，那么 ChooseFont 将返回 TRUE 。随后，POPPAD3 调用 PopFontSetFont 来设置编辑控制中的新字体，旧字体将被删除。

最后，在 WM_DESTROY 消息期间， POPPAD3 调用 PopFontDeinitialize 来删除最近一次由 PopFontSetFont 创建的字体。

11.4.3 查找与替换

通用对话框库也包含两个用于文本查找和替换函数的对话框，这两个函数（ FindText 和 ReplaceText ）使用一个类型为 FINDREPLACE 的结构。图 11-11 中所示的 POPFIND.C 文件有两个例程（ PopFindFindDlg 和 PopFindReplaceDlg ）调用这些函数，还有两个函数在编辑控制中查找和替换文本。

使用查找和替换函数有一些考虑。首先，它们激活的对话框是非模态对话框，这意味着必须改写消息循环，以使在对话框活动时调用 IsDialogMessage 。第二，传送给 FindText 和 ReplaceText 的 FINDREPLACE 结构必须是一个静态变量，因为对话框是非模态的，函数在对话框显示之后返回，而不是在对话框结束之后返回；而对话框过程必须仍然能够访问该结构。

第三，在显示 FindText 和 ReplaceText 对话框时，它们通过一条特殊消息与拥有者窗口通讯，消息号可以通过以 FINDMSGSTRING 为参数调用 RegisterWindowMessage 函数来获得。这是在 WndProc 中处理 WM_CREATE 消息时完成的，消息号存放在静态变量中。

在处理缺省消息时，WndProc将消息变量与RegisterWindowMessage返回的值相比较。lParam 消息参数是一个指向 FINDREPALCE 结构的指针， Flags 字段指示用户使用对话框是为了查找文本还是替换文本，以及是否要终止对话框。 POPPAD3 是调用 POPFIND.C 中的 PopFindFindText 和 PopFindReplaceText 函数来执行查找和替换功能的。

11.4.4 只调用一个函数的 Windows 程序

到现在为止，我们已经给出了两个程序，让你浏览被选颜色，这两个程序分别是第 8 章中的 COLORS1 和本章中的 COLORS2 。现在是给出 COLORS3 的时候了，这个程序只有一个 Windows 函数调用。 COLORS3 的源代码如图 11-13 所示。

COLOR3 所调用的唯一 Windows 函数是 ChooseColor，这也是通用对话框库中的函数，它显示如图 11-14 所示的对话框。颜色选择类似于 COLORS1 和 COLORS2 ，但是它的交互能力更强。

COLORS3.MAK

```
#-----------------------
# COLORS3.MAK make file
#-----------------------

colors3.exe : colors3.obj
     $(LINKER) $(GUIFLAGS) -OUT:colors3.exe colors3.obj $(GUILIBS)

colors3.obj : colors3.c
     $(CC) $(CFLAGS) colors3.c
```

COLORS3.C

```
//*--------------------------------------------
   COLORS3.C - Version using Common Dialog Box
               (c) Charles Petzold, 1996
  --------------------------------------------*/

#include <windows.h>
#include <commdlg.h>

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static CHOOSECOLOR cc ;
     static COLORREF    crCustColors[16] ;

     cc.lStructSize    = sizeof (CHOOSECOLOR) ;
     cc.hwndOwner      = NULL ;
     cc.hInstance      = NULL ;
     cc.rgbResult      = RGB (0x80, 0x80, 0x80) ;
     cc.lpCustColors   = crCustColors ;
     cc.Flags          = CC_RGBINIT | CC_FULLOPEN ;
     cc.lCustData      = 0L ;
     cc.lpfnHook       = NULL ;
     cc.lpTemplateName = NULL ;

     return ChooseColor (&cc) ;
     }
```

图 11-13 COLORS3程序

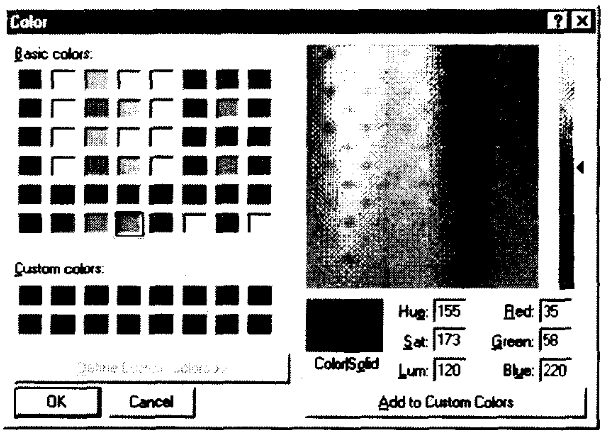


图 11-14 COLORS3的显示

ChooseColor 函数使用一个 CHOOSECOLOR 类型的结构和含有 16 个 COLORREF（ DWORD ）的数组来存放常用颜色，用户将从对话框中选择这些颜色之一。 rgbResult 字段可以初始化为一个颜色值，如果 Flags 字段的 CC_RGBINIT 标志被置位，则显示该颜色。通常在使用这个函数时， rgbResult 将被设置为用户选择的颜色。

注意， Color 对话框的 hwndOwner 字段被设置为 NULL ；在 ChooseColor 函数调用 DialogBox 以显示对话框时， DialogBox 的第三个参数也被设置为 NULL 。这是完全合法的，其含义是对话框不为另一个窗口所拥有。对话框的标题将显示在任务栏中，而对话框就像一个普通的窗口那样运行。

你也可以在自己程序的对话框中使用这种技巧，使 Windows 程序只创建对话框，其它事情都在对话框过程中完成，这是可能的。

第 12 章　现代用户界面

Microsoft Windows 的每个主要版本都会对用户界面加以增强，Windows 95 也不例外，它提供了大量的特色功能来提高使用的方便性。Windows 资源管理器的各种形式（包括网上邻居和控制面板）简化了对硬盘、网络资源以及控制系统设置的窗口的浏览。一个新的外壳紧密地将旧的 MS-DOS 文件系统集成到一个包括网络文件和打印服务器的名字空间中。文件系统中最主要的可见部分是桌面（可以在命令行上用\WINDOWS\DESKTOP 获得），它提供了一种由用户定义的可使用程序和文件的视图。文件系统的另一个可见部分是 Start 菜单（位于\WINDOWS\Start Menu 中），它提供一种层次化的菜单，作为替换桌面的另一种组织程序和数据文件的方式。

为了简化与系统外壳的优美用户界面相配的 Windows 程序的生成，Microsoft 创建了用户控制库。在十七种通用控制中，有一些（如工具栏和状态栏等）已经在许多应用中使用了很多年，这些应用的开发人员都乐于创建一组控制（或者使用由 Microsoft Foundation Class(MFC)库或 Borland 的 OWL 等类库提供的集合）。其它的通用控制，如层次化的树形视图和可配置列表视图，是在 Windows 95 中首次引入的。图 12-1 总结了通用控制，将它们分成四类：框架控制、复合对话控制、Windows 资源管理器控制和杂类控制。

类别	控制	描述
框架窗口控制		常在框架窗口中使用的控制
	工具栏	包含命令快捷按钮
	工具提示	通过在一个浮动窗口中显示文本从而给用户提供快速帮助
	状态栏	信息窗口，通常位于一个应用的框架窗口的底部
复合对话框		用于属性表格和向导(wizard)的控制。
	属性页	非模态对话框，在一个属性表格或者向导中用作单个的页。
	属性表格	多个属性页的容器。
Windows 资源管理器控制		用于建立类似 Windows 资源管理器的用户界面的控制。
	树形视图	显示层次化的组织列表（如 Windows 资源管理器的左半部分）
	列表视图	显示由位图加上文本数据表示的列表项（如 Windows 资源管理器的右半部分）。

续表

类别	控制	描述
杂类控制		
	动画	播放动画序列来显示长时间操作的进展情况。
	拖动列表	支持对自己或者其它拖动列表框进行简单的拖/放。（并非 OLE 的拖/放容器。）
	标题头	为列标记显示一个水平的窗口（由列表视图控制使用）。
	热键	为加速键的运行定义显示反馈信息。
	图象列表	存储一组光栅（位图、光标和图标）图象的非窗口控制。
	进展条	显示长时间操作的进展，以完成任务的百分数表示。
	丰富编辑	一种支持多种字体和基本 OLE 容器功能的编辑控制。
	Tab	显示项目列表以供选择。 Tab 由属性表格用来选择属性页。 Windows 95 任务栏就是一种 Tab 控制，它用按钮代替了 Tab 。
	跟踪条	滚动条的一个替换，用于在特定范围内选择一个值。
	上下控制	滚动条的一个替换，其中包含两个箭头（但是没有滚动条体），用在相应的“伙伴”编辑控制中增大或减小一个值。

图 12-1　COMCTL32.DLL所支持的控制

本章讲述了使用这些控制的基础知识，然后集中讨论前两类——框架窗口控制和复合对话控制，它们代表了每个 Windows 应用都将使用的一组控制；其它的两类—— Windows 资源管理器控制和杂类控制——在 Nancy Cluts 的书《 Windows 95 用户界面程序设计》（ Microsoft Press, 1995 ）中有详细的讨论。

12.1　通用控制基础

每个通用控制——除了图象列表这唯一的一个例外——都被实现为一个窗口类，这样，通用控制类似于 Windows 的第一个版本中引入的预定义对话框控制。这两种类型的控制都是通过调用 CreateWindow 来创建的，并通过使用特定于类的风格标志进行更精确的调整，使用特定于类的消息进行处理，而且通过正规的窗口操纵 API 调用来进行组织。两种类型的控制都向其父窗口发送通知消息以便告诉它们一些感兴趣的事件。

通用控制和预定义控制之间的一个区别在于为通知而发送的特定消息。预定义控制发送 WM_COMMAND 通知消息，而通用控制（除了少数几个例外）发送 WM_NOTIF 消息。虽然发送技术稍微有些不同，但是两种通知消息的思想是一样的：父窗口可能希望响应的某些事情发生了。

当使用其中任何一组中的某个控制时，需要记住的最重要的一点是：控制即窗口，而你所了解的关于操纵窗口的一切都适用于操纵控制。同时还应该记住你已经了解的适用于预定义对话控制的一切也都适用于操纵通用控制。事实上，通用控制之一的丰富编辑控制基本上就是一个增强的编辑控制，它支持由原来的编辑控制所支持的一组基本的控制风格、消息和通知（同时增加了几个为其新的功能特性提供支持）。你已经具有的关于常规编辑控制的知识将有助于你更进一步掌握丰富编辑控制。

和预定义的控制一样，使用通用控制的好处是你可以花费最小的努力使用更多的功能特性——其中每一种都是一个自足的“黑盒子”。使用通用控制的关键是真正理解控制并准确地区分出其固有的功能集和为得到你所希望的工作或者外观，你必须额外进行的工作。一旦你知道了什么是可能的，那么将只需要一点努力就可以让一种控制为你所用。

12.1.1 库初始化

除了丰富编辑控制外，通用控制都在 COMCTL32.DLL 中， COMCTL32.DLL 是在 Microsoft Windows NT3.51 和 Windows 95 中首次发行的（尽管这个 DLL 的名字可能使你觉得有一个 16 位版本的通用控制库是在别的什么地方，但是，到写此书时为止， Microsoft 也没打算发行通用控制库的 16 位版本； Microsoft 确实也曾在 Windows for Workgroup 3.11 中使一个类似的控制库的部分细节可以利用，但这个库从来没有被正式支持过，并且在当前的 Microsoft 操作系统中也不存在）。

为了使用通用控制，程序首先调用 InitCommonControls 函数，该函数使用 RegisterClass 函数注册通用控制窗口类。 InitCommonControls 不需要参数，并且没有返回值：

```
InitCommonControls ( ) ;
```

这个函数的声明，以及需要使用通用控制库的其它声明，都在 COMMCTRL.H 中。这个文件不是在 WINDOWS.H 中引用的核心文件的一部分，因此引用通用控制的函数、数据类型或符号常量的任何一个源文件，都必须显式地包含下列语句行：

```
#include <commctrl.h>
```

为了帮助连接程序定位通用控制函数，必须引用通用控制静态链接库 COMCTL32.LIB ，你的开发环境中可能已经包含了这样的一个引用。你很快将知道，如果缺少这种引用，那么连接程序将显示一条如下所示的消息：

```
error: nuresolved external symbol __imp_InitCommonControls@0
```

为了解决这个问题，在连接程序的库列表中加入 COMCTL32.LIB 。

由于大小和复杂性的关系，丰富编辑控制在它自己的动态链接库 RICHED32.DLL 中。（ Microsoft 没有计划要发布一个 RICHED16.DLL ）丰富编辑控制在装入这个库时注册它自己，装入库所用的函数是 LoadLibrary ：

```
LoadLibrary ("RICHED32.DLL") ;
```

丰富编辑的有关声明在 RICHEDIT.H 中，而特定于 OLE 的丰富编辑控制声明位于 RICHOLE.H 中。

12.1.2 创建通用控制

创建一个通用控制窗口最明显的方法是调用 CreateWindow 或者 CreateWindowEx 函数。（CreateWindowEx 函数和 CreateWindow 相同，只是前者有一个额外的“扩展控制风格”参数。通用控制的这个扩展窗口风格将在下一节中讨论。）例如，下面的调用创建一个工具栏：

```
HWND hwndToolBar = CreateWindow (TOOLVARCLASNAME,
                        NULL,
                        CCS_TOP   |
                        WS_CHILD |
                        WS_VISIBLE |
                        WS_BORDER  |
                        WS_CLIPSIBLINGS,
                        0, 0, 0, 0,
                        hwndParent,
                        (HMENU) ;1,
                        hInst,
                        0) ;
```

这个函数调用中的某些参数值得更深入的解释。类名不出现在引号内，因为它是一个符号常量，其定义依赖于编译连接程序时选择的字符集。对于 ANSI 字符集，TOOLBARCLASSNAME 被替换为“ToolbarWindows32”；而当使用的是 UNICODE 字符集时，将在这个名字前放一个“L”（L“ToolbarWindow32”）从而形成一个 UNICODE 串。所有的通用控制类都采用这种方式定义。

大多数时候，通用控制是作为子窗口创建的，因此使用 WS_CHILD 风格标志和父窗口句柄 hwdParent。在这个例子中，子窗口使用等于零的初始位置(x,y)和大小(cx,cy)，然后当父窗口改变大小（也就是当父窗口获得一条 WM_SIZE 的消息）时，再进行缩放。

另一种调用 CreateWindows 的方式是调用一个特定于控制的创建函数，该函数主要执行某些常见的初始化。一个有关控制的创建函数的例子是 CreateToolbarEx，它创建一个工具栏并在其中增加按钮。在其它情况下，如属性表格等，由于没有类名可用，因此要求一个特定的创建函数：PropertySheet 创建一个属性表格，而 CreatePropertySheetPage 创建单个的属性页。图象列表要求通过调用特定于控制的函数 ImageList_Create 来创建，因为图象列表本身不是窗口。图 12-2 总结了通用控制类名和创建函数。

类别/控制	控制类	控制创建函数
框架窗口控制:		
工具栏	TOOLBARCLASSNAME	CreateToolbarEx
工具提示	TOOLTIPS_CLASS	无
状态栏	STATUSCLASSNAME	CreateStatusWindow
动画	ANIMATE_CLASS	无
进展条	PROGRESS_CLASS	无
复合对话控制:		
属性页	无	CreatePropertySheetPage
属性表格	无	PropertySheet
资源管理器控制:		
树形视窗	WC_TREEVIEW	无
列表视图	WC_LISTVIEW	无
图象列表	无	ImageList_Create
其它控制:		
拖动列表	"Listbox" (ANSI) ;或者 L"Listbox" (UNICODE) ;	MakeDragList
标题头	WC_HEADER	无
热键	HOTKEY_CLASS	无
丰富编辑	"RichEdit" (ANSI) ;或者 L"RichEdit" (UNICODE) ;	无
Tab	WC_TABCONTROL	无
跟踪条	TRACKBAR_CLASS	无
上下控制	UPDOWN_CLASS	CreateUpDownControl

图 12-2 通用控制类名和创建函数

12.1.3 通用控制风格

在创建任何一类窗口（无论是你自己的窗口、预定义的控制，还是通用控制）时，大量的艰苦工作涉及选择正确的风格标志集（复习一下，风格标志使用 C 中的按位或操作符组合在一起，并且作为 CreateWindowEx 中两个参数中的一个：或者是第一个参数 dwExStyle 或者是第四个参数 dwStyle ）。这是一项艰苦的工作，因为风格标志控制广泛的功能特性，包括窗口外观（否则，如果不小心忽略了 WS_VISIBLE，将导致失败）、窗口行为和窗口之间的交互的某些类型。

当创建通用控制时，需要考虑四种风格标志集：一般窗口风格（ WS_）标志、一般通

用控制风格（ CCS_）标志、特定于控制的风格标志和扩展风格（ WS_EX_）标志。前三种类型被传递给 CreateWindowEx 的第四个参数；最后一种类型被传递给 CreateWindowEx 的第一个参数。

12.1.3.1 一般窗口风格

一般窗口风格的名字以 WS_开头，并且可以影响每一种类型的窗口。在大约 20 种窗口风格中，可以应用于通用控制的有： WS_CHILD 、 WS_VISIBLE 、 WS_DISABLED 、 WS_BORDER 、 WS_TABSTOP 、 WS_CLIPCHILDREN 和 WS_CLIPSIBLINGS 。

每个通用控制窗口都将使用 WS_CHILD 风格，从而使通用控制成为在其下方的父窗口的一个子窗口。当控制发送通知消息时，它将这些消息发送给它的父窗口。子窗口在其父窗口被清除时自动被清除。

WS_VISIBLE 风格标志允许一个窗口显示出来（虽然这个标志的出现并不能保证窗口不会被其它窗口所隐蔽）。一个常见的编程错误是忘记包括 WS_VISIBLE 风格标志，而导致程序员疯狂地搜索这个窗口“丢失”的原因——在书里、杂志里和联机帮助里。在创建之后修改一个窗口的可见性可以通过调用 ShowWindow 或者 SetWindowPos 。

WS_DISABLED 标志禁止一个窗口接收鼠标和键盘输入。这个标志最常见的使用是禁止一个对话框中的控制；当被禁止时，大多数的控制将改变它们的外观，让用户知道这些控制是不可使用的。例如，当文件名编辑控制为空时，文件打开控制对话框中的 OK 按钮被禁止。只有当键入某些文本作为文件名时，该按钮才成为可用的。创建窗口后，调用 EnableWindow 函数可以启用或者禁止窗口。

WS_BORDER 风格在控制周围提供一个细小的边框。

当控制位于对话框中时， WS_TABSTOP 风格使该控制成为键盘输入的制表符浏览顺序中的一个停留位置。

WS_CLIPCHILDREN 和 WS_CLIPSIBLINGS 风格保护子窗口的绘制表面不会由于外部干扰而破坏。"Clipping"强调在窗口之间划分边界。虽然对于覆盖和弹出式窗口来说，剪裁(clipping)总是启用的，但是缺省地，剪裁对子窗口是禁止的。当一个子窗口的父窗口被绘制时， WS_CLIPCHILDREN 风格允许对子窗口的剪裁，从而防止了父窗口被绘制在其子窗口上。 WS_CLIPSIBLING 风格允许兄弟窗口之间的剪裁，防止“兄弟相争”——两个兄弟窗口之间争夺重叠区域中的象素的斗争。处理子窗口时，这两种风格有助于解决子窗口外观可能出现的混乱。

12.1.3.2 扩展窗口风格

扩展窗口风格的名字以 WS_EX_开头，并且作为第一个参数传递给 CreateWindowEx 。在 27 个扩展风格位中， 3 个用于子窗口创建： WS_EX_CLIENTEDGE 、 WS_EX_STATICEDGE 和 WS_EX_NOPARENTNOTIFY 。

WS_EX_CLIENTEDGE 和 WS_EX_STATICEDGE 风格标志支持 3D 的外观。适当地使用这些风格允许一个应用比其它 Windows95 应用更好看。注意这些风格标志只在 Windows 95 中和运行 Windows 95 外壳的 Windows NT 的版本中被支持（这些标志对早期的操作系统版本没有效果）。

WS_EX_CLIENTEDGE 风格标志为应用程序的主工作空间创建一个"下陷"的客户区，例如一个字处理程序的文本窗口。由于大多数的控制，如工具栏和状态窗口等，是在这个区域之外创建的，应该避免为工具栏的父窗口和状态窗口的父窗口使用这一风格位。并且，避免使用 WS_EX_OVERLAPPEDWINDOW 风格标志，因为该标志包含了 WS_EX_CLIENTEDGE 作为其定义的一部分。

WS_EX_STATICEDGE 风格标志为只用于输出的窗口创建一种下陷的外观。例如，为 Windows 95 任务栏上的通知窗口——在任务栏右边的小窗口中包含了状态图标和时钟——使用这种风格。你可以为诸如进展条和动画窗口之类的控制使用这一标志。而且，所有嵌入一个状态窗口或者一个工具栏中的仅供输出的控制都可能会使用这一风格标志，因为它的独特外观已经被有经验的 Windows 95 用户所接受。

WS_EX_NOPARENTNOTIFY 风格位要求子窗口不向其父窗口发送 WM_PARENTNOTIF 消息。如果没有这一位，那么一个子窗口将在被创建、被清除或者获得一个鼠标按下消息时向其父窗口发送通知。对话框控制总是用这个风格位来创建的，从而减少了消息通讯的负担。

12.1.3.3 一般通用控制风格

通用控制库提供了一组以一个 CCS_为前缀的风格值，用于工具栏、状态窗口和标题控制。在这一组标志中包括 CCS_ADJUSTABLE、CCS_BOTTOM、CCS_NODIVIDER、CCS_NOMOVEY、CCS_NOPARENTALIGN、CCS_NORESIZE 和 CCS_TOP。由于这些风格的函数是特定于某个控制的，因此更深入的讨论将等到下一节中讨论特定于控制的风格时。

12.1.3.4 特定于控制的风格

就如同预定义对话控制拥有特定于控制的风格——如 BS_PUSHBUTTON、ES_MULTILINE、LBS_SORT 等——一样，Windows 95 的通用控制也是如此。与预定义控制一样，每个通用控制风格都有一个唯一的前缀，所有的前缀被总结在图 12-3 中。更简单的通用控制没有定义特定于控制的风格标志。

关于各个特定于控制的风格的详细讨论将出现在本章后面对各个控制的讨论中。

类别/控制	风格标志前缀	举例
框架窗口控制:		
工具栏	TBSTYLE_	TBSTYLE_ALTDRAG
工具提示	无	
状态栏	SBARS_	SBARS_SIZEGRIP
动画	ACS_	ACS_AUTOPLAY
进展条	无	
复合对话框控制:		
属性页	无	
属性表格	无	

续表

资源管理器控制:		
树视图	TVS_	TVS_HASBUTTONS
列表视图	LVS_	LVS_ALIGNLEFT
图象列表	无	
其它控制:		
拖动列表	无	
标题头	HDS_	HDS_BUTTONS
热键	无	
丰富编辑	ES_	ES_DISABLENOSCROLL
Tab	TCS_	TCS_BUTTONS
跟踪条	TBS_	TBS_AUTOTICKS
上下控制	UDS_	UDS_ALIGNLEFT

图 12-3 通用控制风格标志前缀

12.1.4 向通用控制发送消息

一旦创建了一个通用控制窗口，你就可以发送消息来管理它们的操作。你可以预见，这会涉及到用熟悉的四个参数调用 SendMessage：窗口句柄、消息 ID 、 wParam 值和 lParam 值。正如会有特定于控制的风格一样，也会有特定于控制的消息。

作为调用 SendMessage 的另一种方式，你可以使用在 COMMCTRL.H 中定义的一组 C 语言宏，这些宏接受一组特定于消息的参数，根据需要组合参数值（例如，将两个 short 压缩为一个 lParam ），然后调用 SendMessage 。返回值也可以被强制类型转换，以减少编译器产生的警告，例如，当由 SendMessage 返回的 LRESULT 值不匹配所期望的返回值时。

作为说明这些宏如何简化了向控制发消息的例子，让我们看一看在一个树形视图控制中增加一个项目的 TVM_INSERTITEM 消息。下面是如何使用 SendMessage 发送这个消息：

```
hItem = (HTREEITEM) SendMessage (hwndTV, TVM_INSERTITEM,
                        0, (LPARAM) (LPTV_INSERTSTRUCT) &tvis) ;
```

下面是如何使用 TreeView_InsertItem 宏发送同样的消息：

```
hItem = TreeView_InsertItem (hwndTV, &tvis) ;
```

宏更便于理解（提供了目标窗口类的名字和消息的名字），它只需要键入一半的字符，但是产生同样的结果，因为它被扩展为使用完全相同的类型强制的 SendMessage 调用。如果要在这两者之间进行选择，很难不选择宏调用。

（如果你对使用预定义控制的一组类似的宏感兴趣，那么可以在 WINDOWSX.H 中找到一组宏，虽然这些宏没有在任何一个帮助文件中被明确地说明，但它们是不言自明的，并且非常便于使用。与用于通用控制的宏一样，在 WINDOWSX.H 中的宏将使你的代码易

于编写和阅读。)

虽然这些宏非常有用，但不幸的是， Win32 文件只包括了一半通用控制的宏定义。COMMCTRL.H 只包括下列类的宏定义：动画、标题、列表视图、属性表格（在 PRSHT.H 中）、树形视图和制表符控制。

由于消息宏如此有用，你将在附带的软盘中发现其它通用控制的一组宏，其中包括下列类的宏定义：热键控制、进展条、丰富编辑控制、状态栏、工具栏、工具提示、跟踪条和上下控制。请参考文件\PETZOLD\CHAP12\COMCTHLP.H 。

你还将在附带的软盘中发现一个程序， CTLMACRO ，它列出所有通用控制消息宏。如图 12-4 所示， CTLMACRO 将所有通用控制宏组织为一种层次结构，可以通过一个树形视图控制进行访问。当你定位了你所需要的消息时，按一下 Copy 按钮，将宏拷贝到剪贴板中，这以后，就是简单地将宏粘贴到你喜欢的程序编辑器中。

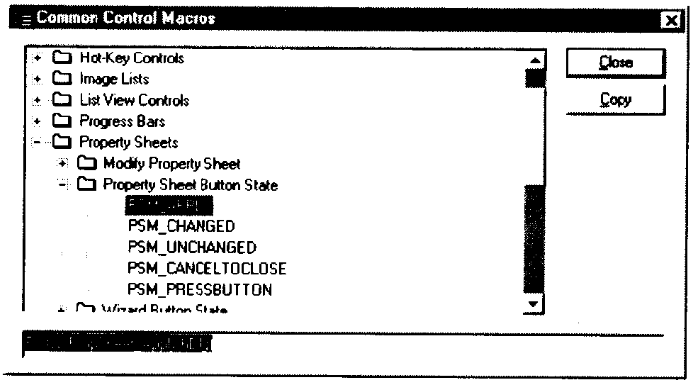


图 12-4 CTLMACRO程序提供对通用控制消息宏的快速访问

12.1.5 来自通用控制的通知消息

和预定义控制一样，通用控制通过通知消息和其父窗口通信。通知消息告诉父窗口在窗口中有父窗口感兴趣的事情发生了：用户单击控制、键入文本、位控制获得了键盘焦点或者焦点从控制中移走了。

预定义控制发送 WM_COMMAND 消息作为通知消息，与此不同，通用控制发送 WM_NOTIFY 消息作为通知消息。因此如果你往现存代码中增加一个通用控制，那么你不太有可能将预定义控制的通知处理和通用控制的通知处理混淆。 WM_NOTIFY 消息还避免了和菜单通知的混淆，菜单通知也是以 WM_COMMAND 消息的形式到达的。

然而，并非所有通用控制通知都以 WM_NOTIFY 消息到达。具体说来就是工具栏，尽管它为大多数的通知使用 WM_NOTIFY ，但在一个工具栏按钮被按下时，它发送 WM_COMMAND 消息。由于工具栏的目的是补充菜单选择，因此，常用的 WM_COMMAND 处理代码——以前支持菜单和加速键——现在可以正确地接受工具按钮消息。另一个例外是上下控制，当按动一个上下按钮时，此控制也发送 WM_VSCROLL 或者 WM_HSCROLL 消息。

虽然每个通用控制有它自己独特的通知码集，仍然定义了一组通用的通知码集。这个

通知码集如下表所示：

通知码	描述
NM_CLICK	用户单击鼠标左键
NM_DBLCK	用户双击鼠标左键
NM_KILLFOCUS	控制失去了键盘焦点
NM_OUTOFMEMORY	内存耗尽错误
NM_RCLICK	用户单击鼠标右键
NM_RDBLCLK	用户双击鼠标右键
NM_RETURN	用户按动 Enter 键
NM_SETFOCUS	控制获得键盘焦点

并非所有通用控制都必须发送所有的通用通知，虽然你也许会认为发送会有意义。例如， Tab 控制不会发送关于焦点的通知（ NM_KILLFOCUS 和 NM_SETFOCUS ）。当你使用通用控制时，你将不得不区别出某种类型的控制发送哪一组通知。

确定特定通用控制类发送什么通知的方法是捕获每个 WM_NOTIFY 消息并将相关的通知码输出到调试窗口。事实上，由于来自每个通用控制的通知码都被定义在一个独立的范围之内，从而简化了这项定位任务。不论你使用的是什么通用控制，你都可以按如下所示的方式处理 WM_NOTIFY 消息，通过 DebugString 函数将输出送给调试字符流。为了看到 Win32 API 中这个函数的结果，你必须在调试器中运行你的程序：

```
case WM_NOTIFY :
     {
     int idCtrl = (int) wParam ;
     LPNMHDR pnmh = (LPNMHDR) lParam ;

     #ifdef _DEBUG
          // Dump notifications to debug stream (debugger)
          LPSTR pText ;
          if (QueryNotifyText (pnmh->code, &pText))
               {
               OutputDebugString (pText) ;
               OutputDebugString ("\r\n") ;
               }
     #endif
     [other program lines]
     return 0 ;
     }
```

本章的每个示例程序都以类似的方式处理 WM_NOTIFY 消息。调用 QueryNotifyText 函数所需要的定义如下所示：

```
typedef struct tagCONTROLNOTIFICATIONS
     {
     UINT  nCode ;
     LPSTR pName ;
```

```
} CONTROLNOTIFICATIONS ;

BOOL QueryNotifyText (UINT nNotifyCode, LPSTR *pName) ;
```

为 QueryNotifText 定义的函数和数据如下所示:

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <prsht.h>
#include "notify.h"

CONTROLNOTIFICATIONS cnLookupTable[] =
     {
     NM_OUTOFMEMORY,           "NM_OUTOFMEMORY",
     NM_CLICK,                 "NM_CLICK",
     NM_DBLCLK,                "NM_DBLCLK",
     NM_RETURN,                "NM_RETURN",
     NM_RCLICK,                "NM_RCLICK",
     NM_RDBLCLK,               "NM_RDBLCLK",
     NM_SETFOCUS,              "NM_SETFOCUS",
     NM_KILLFOCUS,             "NM_KILLFOCUS",
     LVN_ITEMCHANGING,         "LVN_ITEMCHANGING",
     LVN_ITEMCHANGED,          "LVN_ITEMCHANGED",
     LVN_INSERTITEM,           "LVN_INSERTITEM",
     LVN_DELETEITEM,           "LVN_DELETEITEM",
     LVN_DELETEALLITEMS,       "LVN_DELETEALLITEMS",
     LVN_BEGINLABELEDITA,      "LVN_BEGINLABELEDITA",
     LVN_BEGINLABELEDITW,      "LVN_BEGINLABELEDITW",
     LVN_ENDLABELEDITA,        "LVN_ENDLABELEDITA",
     LVN_ENDLABELEDITW,        "LVN_ENDLABELEDITW",
     LVN_COLUMNCLICK,          "LVN_COLUMNCLICK",
     LVN_BEGINDRAG,            "LVN_BEGINDRAG",
     LVN_BEGINRDRAG,           "LVN_BEGINRDRAG",
     LVN_GETDISPINFOA,         "LVN_GETDISPINFOA",
     LVN_GETDISPINFOW,         "LVN_GETDISPINFOW",
     LVN_SETDISPINFOA,         "LVN_SETDISPINFOA",
     LVN_SETDISPINFOW,         "LVN_SETDISPINFOW",
     LVN_KEYDOWN,              "LVN_KEYDOWN",
     HDN_ITEMCHANGINGA,        "HDN_ITEMCHANGINGA",
     HDN_ITEMCHANGINGW,        "HDN_ITEMCHANGINGW",
     HDN_ITEMCHANGEDA,         "HDN_ITEMCHANGEDA",
     HDN_ITEMCHANGEDW,         "HDN_ITEMCHANGEDW",
     HDN_ITEMCLICKA,           "HDN_ITEMCLICKA",
     HDN_ITEMCLICKW,           "HDN_ITEMCLICKW",
     HDN_ITEMDBLCLICKA,        "HDN_ITEMDBLCLICKA",
     HDN_ITEMDBLCLICKW,        "HDN_ITEMDBLCLICKW",
     HDN_DIVIDERDBLCLICKA,     "HDN_DIVIDERDBLCLICKA",
     HDN_DIVIDERDBLCLICKW,     "HDN_DIVIDERDBLCLICKW",
     HDN_BEGINTRACKA,          "HDN_BEGINTRACKA",
     HDN_BEGINTRACKW,          "HDN_BEGINTRACKW",
     HDN_ENDTRACKA,            "HDN_ENDTRACKA",
     HDN_ENDTRACKW,            "HDN_ENDTRACKW",
     HDN_TRACKA,               "HDN_TRACKA",
     HDN_TRACKW,               "HDN_TRACKW",
```

```
     TVN_SELCHANGINGA,      "TVN_SELCHANGINGA",
     TVN_SELCHANGINGW,      "TVN_SELCHANGINGW",
     TVN_SELCHANGEDA,       "TVN_SELCHANGEDA",
     TVN_SELCHANGEDW,       "TVN_SELCHANGEDW",
     TVN_GETDISPINFOA,      "TVN_GETDISPINFOA",
     TVN_GETDISPINFOW,      "TVN_GETDISPINFOW",
     TVN_SETDISPINFOA,      "TVN_SETDISPINFOA",
     TVN_SETDISPINFOW,      "TVN_SETDISPINFOW",
     TVN_ITEMEXPANDINGA,    "TVN_ITEMEXPANDINGA",
     TVN_ITEMEXPANDINGW,    "TVN_ITEMEXPANDINGW",
     TVN_ITEMEXPANDEDA,     "TVN_ITEMEXPANDEDA",
     TVN_ITEMEXPANDEDW,     "TVN_ITEMEXPANDEDW",
     TVN_BEGINDRAGA,        "TVN_BEGINDRAGA",
     TVN_BEGINDRAGW,        "TVN_BEGINDRAGW",
     TVN_BEGINRDRAGA,       "TVN_BEGINRDRAGA",
     TVN_BEGINRDRAGW,       "TVN_BEGINRDRAGW",
     TVN_DELETEITEMA,       "TVN_DELETEITEMA",
     TVN_DELETEITEMW,       "TVN_DELETEITEMW",
     TVN_BEGINLABELEDITA,   "TVN_BEGINLABELEDITA",
     TVN_BEGINLABELEDITW,   "TVN_BEGINLABELEDITW",
     TVN_ENDLABELEDITA,     "TVN_ENDLABELEDITA",
     TVN_ENDLABELEDITW,     "TVN_ENDLABELEDITW",
     TVN_KEYDOWN,           "TVN_KEYDOWN",
     TTN_NEEDTEXTA,         "TTN_NEEDTEXTA",
     TTN_NEEDTEXTW,         "TTN_NEEDTEXTW",
     TTN_SHOW,              "TTN_SHOW",
     TTN_POP,               "TTN_POP",
     TCN_KEYDOWN,           "TCN_KEYDOWN",
     TCN_SELCHANGE,         "TCN_SELCHANGE",
     TCN_SELCHANGING,       "TCN_SELCHANGING",
     TBN_GETBUTTONINFOA,    "TBN_GETBUTTONINFOA",
     TBN_GETBUTTONINFOW,    "TBN_GETBUTTONINFOW",
     TBN_BEGINDRAG,         "TBN_BEGINDRAG",
     TBN_ENDDRAG,           "TBN_ENDDRAG",
     TBN_BEGINADJUST,       "TBN_BEGINADJUST",
     TBN_ENDADJUST,         "TBN_ENDADJUST",
     TBN_RESET,             "TBN_RESET",
     TBN_QUERYINSERT,       "TBN_QUERYINSERT",
     TBN_QUERYDELETE,       "TBN_QUERYDELETE",
     TBN_TOOLBARCHANGE,     "TBN_TOOLBARCHANGE",
     TBN_CUSTHELP,          "TBN_CUSTHELP",
     UDN_DELTAPOS,          "UDN_DELTAPOS",
     PSN_SETACTIVE,         "PSN_SETACTIVE",
     PSN_KILLACTIVE,        "PSN_KILLACTIVE",
     PSN_APPLY,             "PSN_APPLY",
     PSN_RESET,             "PSN_RESET",
     PSN_HELP,              "PSN_HELP",
     PSN_WIZBACK,           "PSN_WIZBACK",
     PSN_WIZNEXT,           "PSN_WIZNEXT",
     PSN_WIZFINISH,         "PSN_WIZFINISH",
     PSN_QUERYCANCEL,       "PSN_QUERYCANCEL"
     } ;

int NOTIFY_COUNT = sizeof (cnLookupTable) /
                   sizeof (CONTROLNOTIFICATIONS) ;
```

```
//----------------------------------------------------------
// QueryNotifyText: Convert notification codes into text
//----------------------------------------------------------
BOOL QueryNotifyText (UINT nNotifyCode, LPSTR *pName)
     {
     BOOL bFound = FALSE ;
     int  iNotify ;

     for (iNotify = 0 ; iNotify < NOTIFY_COUNT ; iNotify++)
          {
          if (cnLookupTable[iNotify].nCode == nNotifyCode)
               {
               *pName = cnLookupTable[iNotify].pName ;
               return TRUE ;
               }
          }

     // Unknown notification code
     *pName = "** Unknown **" ;
     return FALSE ;
     }
```

12.2 框架窗口控制

有三个通用控制经常用于框架窗口：工具栏、工具提示和状态栏，图 12-5 显示了 GADGETS 示例程序中实现的这三种控制。

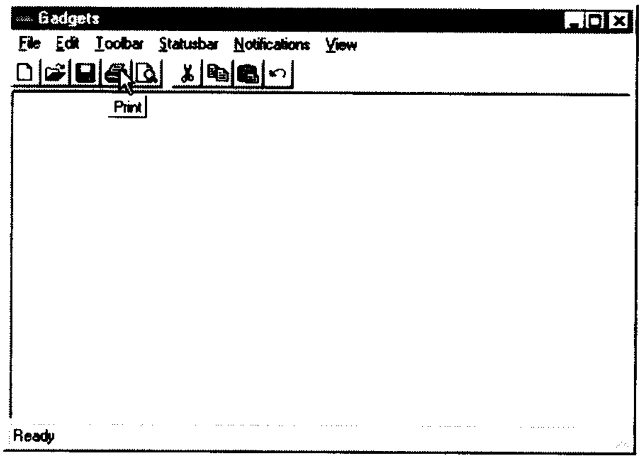


图 12-5 GADGETS程序，显示了工具栏、工具提示和状态栏框架窗口通用控制

12.2.1 工具栏

一个工具栏是一个子窗口，一般正好位于程序菜单的下面，其中包含经常要用的菜单命令和程序选项所对应的按钮。工具栏按钮本身不是窗口，而是使用位图绘制在工具栏窗

口表面的图形对象。

工具栏按钮上的标记可以是位图，也可以是带有补充的文本标记的位图（当前的实现不支持仅有文本标记的按钮）。工具栏将所有按钮的大小设置为相同的，并将带有文本标记的按钮的大小设置得足够大以容纳最长的标记。由于这个原因，你很可能希望使用短字符串作为文本标记以避免过大的按钮。

除了按钮外，工具栏还可以容纳其它子窗口控制，如组合框等。使用 CreateWindow 调用创建被包含的控制，指定工具栏作为被包含控制的父窗口。如 Nancy Cluts 在《 Windows 95 用户界面程序设计》中所指出的，将控制放在工具栏上的真正障碍是保留足够的象素空间。这是通过增加分隔符占据被包含控制的位置来实现的。正如菜单中的分隔符在菜单定义中是使用一种特殊标志 SEPARATOR 关键字来定义的一样，工具栏按钮分隔符的创建是通过使用 TBSTYLE_SEP 按钮分隔标志来创建工具栏按钮而完成的。要为工具栏中的组合框保留足够空间，需要大约一打的分隔符。

12.2.2 创建工具栏

创建工具栏，可以通过调用 CreateWindow 并指定 TOLLBARCLASSNAME 类名，也可以通过调用 CreateToolbarEx ，它创建工具栏并初始化一组按钮。下面所示是 CreateToolbarEx 函数的定义：

```
HWND CreateToolbarEx (HWND hwnd, DWORD ws, UINT wID,
          int nBitmaps, HINSTANCE hBMInst,
          UINT wBMID, LPCTBBUTTON lpButtons,
          int  iNumButtons, int dxButton,
          int dyButton, int dxBitmap,
          int dyBitmap, UINT uStructSize) ;
```

前三个参数用于 CreateToolbarEx 对 CreateWindow 的调用中： hwnd 是父窗口， ws 包含窗口风格标志，而 wID 是工具栏子窗口的 ID 。

后三个参数用于装入位图资源，这些位图包含按钮的表面图象（多个图象被压缩在同一个位图中的一行里）： nBitmaps 是位图中的图象个数，而 hBMInst 和 wBMID 标识要装入的位图资源。

lpButton 是指向成员为 TBBUTTON 的数组的指针。每个 TBBUTTON 成员定义每个工具栏按钮的位图、命令 ID 、按钮类型和初始按钮状态。

每个按钮的大小基于位图图象的大小（ dxBitmap, dyBitmap ）进行设置。一个按钮的最小宽度是 dxBitmp + 7 个象素，而最小高度是 dyBitmap+7 个象素。你可以选择设置 dxButton 和 dyButton ，强制一个按钮大于这个最小值；否则，将 dxButton 和 dyButton 设置为 0 。

工具栏的版本基于 TBBUTTON 结构的大小，你在最后一个参数—— uStructSize ——中指定，它必须被设置为 sizeof(TBBUTTON)。

在创建时，工具栏将它的大小和位置设置为“精确”值：高度被设置为容纳它的按钮，宽度被设置为其父窗口的客户区的宽度，而位置被设置在其父窗口的顶部。要想让工具栏

以不同于此的方式显示则要求修改它的窗口风格，这将在下一节中讨论。

12.2.2.1 工具栏窗口风格

工具栏的主要特征通过设置工具栏窗口风格标志来控制，允许一般（CCS_）风格标志和特定于工具栏的（TBSTYLE_）风格标志的混合使用，如下表中所示：

类别	风格标志	描述
外观	CCS_NODIVIDER	禁止在工具栏的上边界的加亮绘制
	TBSTYLE_WRAPABLE	支持多行工具栏
在 y 轴上的自动移动	CCS_TOP	将工具栏放置在父窗口的顶部（缺省），并将宽度调节为父窗口宽度，高度调节为按钮大小
	CCS_BOTTOM	将工具栏放置在父窗口的底部，并将宽度调节为父窗口客户区的宽度，高度调节为按钮的大小
	CCS_NOMOVEY	设置初始 x 位置（在父窗口的左边）而非初始 y 位置，将宽度调节为父窗口客户区的宽度，高度调节为按钮的大小
禁止自动移动/自动缩放	CCS_NOPARENTALIGN	工具栏设置自己的高度，但不设置位置和宽度。要想得到期望的结果，在创建工具栏之后发送一条缩放消息
	CCS_NORESIZE	禁止所有的自动移动和自动缩放功能特性，这禁止了下列风格标志：CS_TOP、CCS_BOTTOM、CCS_NOMOVEY 和 CCS_NOPARENTALIGN。你必须明确地移动和缩放工具栏
工具栏定制	CCS_ADJUSTABLE	支持 Shift+单击来移动按钮以及双击来显示定制的对话框。（细节请参见后面关于定制工具栏的讨论。）
	TBSTYLE_ALTDRAG	修改 CCS_ADJUSTABLE 工具栏，使用 Alt+单击来移动按钮
工具提示支持	TBSTYLE_TOOLTIPS	创建一个工具提示控制。

唯一一个缺省风格标志是 CCS_TOP，该标志在创建工具栏时，将工具栏设置在其父窗口的客户区的顶部。这以后，父窗口可以通过发送一个 TB_AUTOSIZE 消息来要求工具栏进行缩放和重定位，工具栏父窗口通常在收到一条 WM_SIZE 消息时才这么做。除了发送缩放请求消息外，带有 CCS_TOP 和 CCS_BOTTON 标志的工具栏能自给自足，而不需要别的维护。如果出现禁止一个工具栏的自动移动和自动缩放特征的风格标志，那么当工具栏的父窗口进行缩放时，需要更多的一些工作来维护工具栏。

有两个风格标志用于修改工具栏的外观。首先是 CCS_NODIVIDER ，它取消将 CCS_TOP 风格工具栏上按钮和应用程序菜单分隔开的一个加亮边框。对于在其它位置绘制的工具栏（如在第一个工具栏下绘制的另一个工具栏），你可能希望隐藏边框；对具有工具栏而没有菜单的窗口存在同样的情况——边框看起来不太合适。另一个外观修改风格是 TBSTYLE_WRAPABLE ，它允许工具栏将按钮分配在多行上。如果没有这个标志，那么太长而无法在单行中容纳的按钮将从视图中被截掉，用户将无法使用它们。

三个风格标志控制 y 轴上的移动： CCS_TOP （缺省）、 CCS_BOTTOM 和 CCS_NOMOVEY 。这些标志控制工具栏在创建时以及收到 TB_AUTOSIZE 消息时，如何在其父窗口中进行定位。 CCS_NOMOVEY 标志使工具栏只调节其宽度（以适应其父窗口的客户区），而其高度和在 x 轴上的位置由你来决定，你可以在 CreateWindow 调用中使用所需要的 y 轴坐标值，或者在工具栏被创建之后再调用 MoveWindow 这样的函数。带有这个标志的工具栏可以用作 CCS_TOP 工具栏之下的另一个工具栏。

其它两个风格标志限制工具栏的自动移动和自动缩放特性。使用 CCS_NOPARENTALIGH 风格标志，工具栏设置它自己的高度以容纳其中按钮的大小——但是你必须设置工具栏的位置和宽度。 CCS_NORESIZE 标志完全禁止对工具栏的任何移动或者缩放。这个标志在使用工具栏作为对话框控制，也即当它必须在不进行缩放的条件下适应某个空间时有用。这个标志在把两个或多个工具栏设置在同一行上时也有用。

CCS_ADJUSTABLE 风格标志创建用户可以任意修改的工具栏。按钮可以沿工具栏拖动，还可以使用 Shift+单击被拖动到工具栏之外。双击操作将弹出用户工具栏对话框，用于增加、删除和移动工具栏按钮（这个对话框还将在工具栏收到 TB_CUSTOMIZE 消息时出现）。定制工具栏要求父窗口对由工具栏发送的几个通知消息作出响应，其中包括 TBN_QUERYINSERT 和 TBN_QUERYDELETE ，它们在插入或删除按钮时请求授权。对于定制的工具栏， TBSTYLE_ALTDRAG 标志将按钮移动方式由 Shift+单击改变为 Alt+单击，而 Shift+单击将有别的含义。

TBSTYLE_TOOLTIPS 风格标志要求工具栏创建工具提示控制，它为每个按钮显示一个小窗口的文本提示。当工具提示需要针对特定按钮的文本时，工具栏的父窗口将获得一个 TTN_NEEDTEXT 通知（以 WM_NOTIFY 消息的形式到达）。

12.2.2.2 定义按钮表面图象

工具栏窗口过程遵循下面的限制将单个位图分解为按钮表面所需要的多个图象，所有的图象必须是相同大小的。具有多于一个图象的位图必须将图象放在同一行中——也就是说，第二个图象出现在第一个图象的右边，第三个出现在第二个的右边，等等。虽然 CreateToolbarEx 函数只接受位图标识符，但是你可以通过向工具栏发送 TB_ADDBITMAP 消息，从而在一个工具栏中增加位图。每个位图可以有一个或者多个按钮图象。

通用控制库有两组预定义的按钮图象位图。一组位图包含对应于 File 和 Edit 菜单的按钮图象；第二组包含不同类型视图的按钮图象。每组以两种大小提供位图：大的（ 24 × 24 ）和小的（ 16 × 16 ）。要访问这些位图，你需要指定一个特定的值， HINTS_COMMCTRL ，作为 CreateToolbarEx 的位图资源实例句柄（ hBMInst ）参数（或者将该参数包含在 TB_ADDBITMAP 消息的适当结构中）。对于位图 ID 参数， wBMID ，

从下表中选择：

位图资源 ID	描述
IDB_STD_SMALL_COLOR	File 和 Edit 菜单图象， 16x16 按钮图象
IDB_STD_LARGE_COLOR	File 和 Edit 菜单图象， 24x24 按钮图象
IDB_VIEW_SMALL_COLOR	View 菜单图象， 16x16 按钮图象
IDB_VIEW_LARGE_COLOR	View 菜单图象， 24x24 按钮图象

两个“标准”（IDB_STD_SMALL_COLOR 和 IDB_STD_LARGE_COLOR）位图每一种都包含 15 个图象。将某个特定图象连接到工具栏按钮涉及在按钮对应的 TBBUTTON 结构的 iBitmap 成员中设置从 0 开始的图象索引。当使用标准位图时，从下面的一组符号常量中选择一个索引：STD_CUT、STD_COPY、STD_PASTE、STD_UNDO、STD_REDRAW、STD_DELETE、STD_FILENEW、STD_FILEOPEN、STD_FILESAVE、STD_PRINTPRE、STD_PROPERTIES、STD_HELP、STD_FIND、STD_REPLACE 和 STD_PRINT。

两个“视图”（IDB_VIEW_SMALL_COLOR 和 IDB_VIEW_LARGE_COLOR）位图每一个都包含 12 个图象。使用下列符号常量选择按钮图象索引：VIEW_LARGEICONS、VIEW_SMALLICONS、VIEW_LIST、VIEW_DETAILS、VIEW_SORTNAME、VIEW_SORTSIZE、VIEW_SORTDATE、VIEW_SORTTYPE、VIEW_PARENTFOLDER、VIEW_NETCONNECT、VIEW_NETDISCONNECT 和 VIEW_NEWFOLDER。

当使用这四组位图中的一个时，你不需要设置按钮的大小（dxButton, dyButton）和位图的大小（dxBitmap, dyBitmap），因为 CreateToolbarEx 认识这些位图并能设置大小。

12.2.2.3 填充TBBUTTON数组

使用位图创建工具栏决定了定义单个工具栏按钮的步骤。首先，你需要一个 TBBUTTON 成员组成的数组，每个成员定义单个工具栏按钮。当创建了一个工具栏后，你可以通过向工具栏发送 TB_ADDBUTTON 消息，或者通过使用 TB_INSERTBUTTON 消息在现有的按钮之间插入按钮，从而定义增加的按钮。

TBBUTTON 在 COMMCTRL.H 中被定义为：

```
typedef struct _TBBUTTON
     {
     int iBitmap ;
     int idCommand ;
     BYTE fsState ;
     BYTE fsStyle ;
     BYTE bReserved[2] ;
     DWORD dwData ;
     int iString ;
     } TBBUTTON ;
```

注意这个定义不同于 Win32 帮助数据库中给出的那一个，以前那个没有包括 bReserved 成员。当执行对 TBBUTTON 成员的块初始化时，你将必须包括两个额外的成员以适当地初始化数组。

TBBUTTON 的 iBitmap 成员是要使用的以 0 为起点的按钮图象的索引。当使用通用控制库提供的位图时，你可以在这个成员中使用 STD_和 VIEW_常量。一个特定的工具栏根据位图被加入的顺序来定义它自己的图象索引。

TBBUTTON 的 idCommand 标识一个按钮的命令标识符。当按钮被按下时，它发送 WM_COMMAND 消息，消息的 wParam 参数被设置为 idCommand 。

fsState 和 fsStyle 指示按钮的初始状态和风格。“按钮风格”在按钮的生命周期中是永久性的；“按钮状态”可以在按钮创建和修改时由用户动作或者由发送给工具栏的消息来设置。下面是五个工具栏按钮风格：

按钮风格	描述
TBSTYLE_BUTTON	按钮动作如同压入按钮（可以按动但是不会保持压下状态——缺省值）
TBSTYLE_SEP	在按钮之间形成空间或者为子窗口控制（如组合框）保留空间的分隔符
TBSTYLE_CHECK	按钮动作如同复选框（每次按动时，在突起和压下之间切换）
TBSTYLE_CROUP	按钮作为类似于单选按钮组的成员（支持互斥选择）
TBSTYLE_CHECKGROUP	组合 TBSTYLE_CHECK 和 TBSTYLE_GROUP 。

五种风格相互组合大致形成了三种基本类型的对话框按钮的等价物：压入按钮、复选框和单选钮。

有六种工具栏按钮状态：

按钮状态	描述
TBSTATE_CHECKED	复选框按钮处于压下状态
TBSTATE_PRESSED	任何类型的按钮处于压下状态
TBSTATE_ENABLED	按钮被启用（可以接收鼠标输入）
TBSTATE_HIDDEN	按钮被隐蔽（完全消失，其空间被其它按钮占据）
TBSTATE_INDETERMINATE	按钮灰化但是仍然能够被按动。
TBSTATE_WRAP	在 TBSTYLE_WRAPABLE 工具栏中，工具栏在这个按钮之后换行

这些状态中的每一种都可以通过对工具栏发出特定的消息来访问：一条消息查询状态而另一条消息设置状态。例如，你可以通过发送 TB_ISBUTTONENABLED 消息来查询指定按钮的允许状态；发送 TB_ENBALEBUTTON 消息来允许或者禁止一个按钮。

TBBUTTON 的 dwData 成员是可供你使用的一个可选的 4 字节域。例如，你可以在这个成员中存储一个指针以容纳特定于按钮的数据。一旦被设置了，你可以通过发送

TB_GETBUTTON 消息来查询 dwData 的值，但是一旦被创建了之后，你就无法改变一个按钮的 dwData 值。

TBBUTTON 的 iString 成员是按钮标记的索引，从 0 开始。你通过发送 TB_ADDSTRING 消息在工具栏的字符串中增加新串。

12.2.2.4 创建工具栏的例子

下面是创建工具栏的例子，工具栏中包括采用由通用控制库提供的标准位图的小型（16x16）按钮表面图象。

```
HWND hwndToolBar = CreateToolbarEx (hwndParent,
                        WS_CHILD | WS_VISIBLE |
                        WS_CLIPSIBLINGS | CCS_TOP |
                        TBSTYLE_TOOLTIPS,
                        1, 0,
                        HINST_COMMCTRL,
                        IDB_STD_SMALL_COLOR,
                        tbb,
                        5, 0, 0, 0, 0,
                        sizeof (TBBUTTON)) ;
```

CreateToolbarEx 的十三个参数允许创建工具栏子窗口，装入单个位图资源作为按钮表面图象的来源，创建五个工具栏按钮，请求工具提示支持，并建立所希望的工具栏窗口版本。

下面是定义单个工具栏按钮所要求的 TBBUTTON 数组：

```
TBBUTTON tbb[] =
    {
    STD_FILENEW,   1, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
    STD_FILEOPEN,  2, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
    STD_FILESAVE,  3, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
    STD_PRINT,     4, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
    STD_PRINTPRE,  5, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, 0,
    } ;
```

如果不调用 CreateToolbarEx，那么你可以调用 CreateWindow 来创建工具栏，然后发送消息来初始化工具栏的各个不同方面。下面的代码产生和前面的 CreateToolbarEx 调用相同的结果：

```
HWND hwndToolBar = CreateWindow (TOOLBARCLASSNAME,
                        NULL,
                        WS_CHILD | WS_VISIBLE |
                        WS_CLIPSIBLINGS | CCS TOP |
                        TBSTYLE_TOOLTIPS,
                        0, 0, 0, 0,
                        hwndParent,
                        (HMENU) 1,
                        hInst,
                        0) ;
```

```
// Set version with TBBUTTON size
ToolBar_ButtonStructSize (hwnd) ;

ToolBar_AddBitmap (hwnd, 1, &tbbitmap) ;

// Create buttons
ToolBar_AddButtons (hwnd, 5, tbb) ;
```

唯一要求的额外的数据定义是一个类型为 TBADDBITMAP 的结构，它标识工具栏位图的实例句柄和资源标识符：

```
TBADDBITMAP tbbitmap =
     {
     HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
     };
```

创建一个工具栏后，每当其中的一个按钮被按动时，它就给其父窗口发送一条 WM_COMMAND 消息。除了对这一类消息的响应外，工具栏控制所需要的全部其它维护工作是要求它在父窗口改变大小时相应地改变它的大小。

12.2.2.5 移动和缩放工具栏

工具栏有一种自动缩放和自动定位的特性，当你发送给工具栏一条 TB_AUTOSIZE 消息时，这种特性将发挥作用。通常你会在工具栏的父窗口改变大小时发送这个消息——即，在收到一条 WM_SIZE 的消息时：

```
SendMessage (hwndToolBar, TB_AUTOSIZE, 0, 0L) ;
```

另一种选择是，你可以使用在附带的软盘上的 COMCTHLP.H 中提供的消息宏：

```
ToolBar_AutoSize (hwndToolBar) ;
```

工具栏对这个消息的响应依赖于它的风格标志。例如，在 CCS_TOP 和 CCS_BOTTOM 风格下，工具栏设置它自己的大小和位置。在另一个极端，带有 CCS_NORESIZE 标志的工具栏将忽略这个消息，而要求你明确地设置它的大小和位置。

12.2.2.6 支持工具提示控制

工具提示控制是一个包含帮助文本的小窗口。一般地，工具提示控制在鼠标光标暂停在一个特定区域，如工具栏按钮上时被激活；不过，也可以随时激活一个工具提示。工具提示控制跟踪一些“热”区域——在讨论工具提示时，称为“工具”——这种区域可以是窗口中的矩形区域，或者是整个窗口。工具提示可以针对位于任何类型窗口中的工具而出现，尽管我们主要关注位于工具栏内的工具提示。

使用一个 TBSTYLE_TOOLTIPS 风格标志创建的工具栏拥有工具提示控制，而每个加入工具栏中的按钮都被注册为工具提示工具。要修改工具栏的工具提示控制——例如，要在工具列表中增加一个组合框控制——先通过给工具栏发送一条 TB_GETOOLTIPS 消息

获取工具提示窗口句柄，然后通过直接向工具提示控制发送工具提示控制消息（以 TTM_前缀作为标记）来修改工具提示控制。

当工具提示在工具栏上被激活时，工具提示通过发送给工具栏一条带有TTN_NEEDTEXT 通知码的 WM_NOTIFY 消息，从而发送一个显示文本的请求。工具栏将这个消息转送给工具栏的父窗口，而父窗口通过填充一个指定的 TOOLTIPTEXT 结构作为响应。下面所示是如何为工具栏窗口中的三个工具栏按钮和一个组合框响应这些消息：

```
case WM_NOTIFY :
     {
     LPNMHDR pnmh = (LPNMHDR) lParam ;
     LPSTR   pReply ;

     // Fetch tooltip text
     if (pnmh->code == TTN_NEEDTEXT)
          {
          LPTOOLTIPTEXT lpttt = (LPTOOLTIPTEXT) lParam ;
          switch (lpttt->hdr.idFrom)
               {
               case 0 :
                    pReply = "First Button" ;
                    break ;

               case 1 :
                    pReply = "Second Button" ;
                    break ;

               case 2 :
                    pReply = "Third Button" ;
                    break ;

               default :
                    if (lpttt->uFlags & TTF_IDISHWND) &&
                        lpttt->hdr.idFrom == (UINT) hwndCombo)
                         {
                         pReply = "Combo Box" ;
                         }
                    else
                         {
                         pReply = "Unknown" ;
                         }
               }
          lstrcpy (lpttt->szText, pReply) ;
          }
     }
```

在这个代码段中有一个奇怪的特点，即 WM_NOTIFY 的 lParam 参数被强制为两种不同的指针类型：一个指向 NMHDR 的指针和一个指向 TOOLTIPTEXT 的指针。这可以通过下面的事实来解释：所有 WM_NOTIFY 消息都传递一个包含适用于每种类型通用控制的细节的 NMHDR 结构。当必须提供特定于控制（或者特定于通知码）的细节时，在 NMHDR 数据后面提供的数据——这里是 TOOLTIPTEXT——以一个 NMHDR 结构作为它的第一个

数据成员。

WINUSER.H 定义的 NMHDR 如下所示：

```
typedef struct tagNMHDR
    {
    HWND  hwndFrom;
    UINT    idFrom;
    UINT    code;
    } NMHDR;

typedef NMHDR FAR *LPNMHDR;
```

这个结构的第三个成员 code 中包含通知码——即 WM_NOTIFY 消息被发送的原因——并且它可以是一个一般的通知码（像 NM_CLICK 或者 NM_SETFOCUS ）或者是一个特定于类的码(如本例中的 TTN_NEEDTEXT 等)。 hwndFrom 成员标识发送消息的窗口， idFrom 可以是窗口 ID 或者发送消息的特定项目（如一个工具栏按钮等）的 ID 。

当工具提示发送 TTN_NEEDTEXT 通知时，它传递一个指向唯一于这个通知码的数据结构的指针： TOOLTIPTEXT 。这个结构在 COMCTRL.H 中定义如下：

```
typedef struct tagTOOLTIPTEXTA
    {
    NMHDR       hdr ;
    LPSTR       lpszText ;
    char        szText[80] ;
    HINSTANCE hinst ;
    UINT        uFlags ;
    } TOOLTIPTEXTA, FAR *LPTOOLTIPTEXTA ;
```

第一个成员 hdr 包含必须为每个 WM_NOTIFY 消息提供的 NMHDR 数据。

当收到 TTN_NEEDTEXT 消息时，第一项任务是检查哪个按钮（或者窗口）被选择。这是通过检查 NMHDR 结构中的 idFrom 成员来完成的，该成员在鼠标位于一个按钮上时，可以是按钮的索引，或者当鼠标位于工具栏的一个子窗口上时，可以是一个窗口句柄。uFlags 成员中包含标志 TTF_IDISHWND ，指示标识符是一个窗口句柄而非按钮的索引。

当收到 TTN_NEEDTEXT 消息时，有三种方法为工具提示控制提供文本：通过返回指向 lpszText 成员中字符串的指针，通过将字符串拷贝到 szText 缓冲区中，或者通过指定一个字符串资源。和对其它资源的处理一样，你使用一个实例句柄和一个唯一的标识符来标识字符串资源。使用 MAKEINTRESOURCE 宏，将实例句柄拷贝给 hinst 成员，将标识符拷贝给 lpszText 成员：

```
lpttt->hinst = hInstance;
lptt->lpszText = MAKEINTRESOURCE (100) ;
```

12.2.2.7 在工具栏上增加子窗口

工具栏只支持按钮，因此要在工具栏上使用非按钮的控制，你必须创建一个子窗口。其中比较常用的窗口类型是组合框；因此，组合框将是这里讨论的重点（虽然任何类型的

窗口都可以在工具栏上使用）。由于我们已经在第 8 章中介绍了创建子窗口的基本知识，因此我将在这里讨论一些更深入的问题：在工具栏上为一个子窗口准备空间、缩放工具栏并提供工具提示支持。

12.2.2.8　在工具栏中为子窗口准备空间

在《Windows 95 用户界面程序设计》中，Nancy Cluts 建议在工具栏中创建一个子窗口时，加入分隔符（带有风格 TBSTYLE_SEP 的按钮）占据位置。你必须通过尝试确定出分隔符的正确数目，作为借鉴，可以参考本章后面出现的 GADGETS 示例程序，它使用了 20 个分隔符为其组合框分配空间。

使用分隔符有利于计算窗口的坐标，这些坐标是你在调用 CreateWindow 时所需要的。发送 TB_GETITEMRECT 消息可以获取工具栏中任何一项——按钮或者分隔符——的象素坐标，并可以获取该项目所在矩形的四个坐标。下面是 GADGETS 如何使用 COMCTHLP.H（可以在附带的软盘中找到）中的消息宏计算可用空间：

```
RECT r ;
int x, y, cx, cy ;

// Calculate coordinates for combo box
ToolBar_GetItemRect (hwndTB, 0, &r) ;
x = r.left ;
y = r.top ;
ToolBar_GetItemRect (hwndTB, 18, &r) ;
cx = r.right - x + 1 ;
```

将控制的左上角设置为(x, y)，从而使控制和工具栏按钮排列在一行中。由于 GADGETS 在其工具栏上放置了 20 个分隔符，因此查询第 19 个（或者，如果从 0 开始数的第 18 个）分隔符的位置将返回组合框右边界和第一个按钮之间的那个分隔符的宽度。

在工具栏上放置一个子窗口唯一需要的另外一个值是高度，对大多数控制来说，这个值可能就是由 TB_GETITEMRECT 消息返回的按钮高度（r.button - r.top +1）。但是组合框和其它类型的窗口稍微有些不同，因为它的编辑（或者静态）控制可以缩放它本身以容纳它的字体，而传递给 CreateWindows 的高度只是其下拉式窗口可允许的高度。设置一个足够大的值以容纳几行是有意义的，你采用一种设备无关的方式来做到这一点，即可以根据由 GetTextMetrics 函数返回的字体尺寸数据来进行计算。GADGETS 将这个值硬编码为：

```
cy = 100;
```

12.2.2.9　创建工具栏控制

当你在工具栏上创建一个子窗口时，存在一些限制。有些是明显的，如确认包含了 WS_VISIBLE 和 WS_CHILD 风格标志。如果没有 WS_VISIBLE，那么调用可能会成功，但是控制却不会出现。WS_CHILD 风格标志创建一个子窗口，但是必须保证父窗口存在，才能在其象素上放置子窗口。从而引发了一个问题，即哪个窗口应该是子窗口的父窗口。

对工具栏窗口来说，最明显的父窗口选择是工具栏自身，但是这可能会引起一些很奇怪的问题。具体说，预定义的控制向其父窗口发送通知消息（以 WM_COMMAND 消息的形式），这些消息将产生不可预料的后果。例如，当 GADGETS 程序的工具栏组合框以工具栏作为其父窗口而被创建，并且组合框窗口的 ID 被设置为 101 时，该程序将在 Windows 95 上不可避免地发生崩溃。（很奇怪，这种组合在 Windows NT 3.51 上却从来没有导致过任何问题。通过仔细的研究，揭示出问题是由于 COMCTL32.DLL 特定于操作系统的版本所引起的。）虽然改变控制的 ID 似乎解决了这个问题，但是仍然存在一种可能，即通用控制库将来的一个版本将使用一个我们选择的一个子窗口 ID ，从而再次导致这个问题的产生。与此同时，工具栏又必须是窗口的父窗口，才能使子窗口位于工具栏之上。

这是一个明显的冲突，解决方案是在调用 CreateWindow 中指定另一个窗口作为子窗口的父窗口——选择一个你拥有其窗口过程的窗口。当子窗口被创建之后，调用 SetParent ，将子窗口放在工具栏上。对话控制显然非常忠实地将它们的 WM_COMMAND 通知消息都发送给它们最初的父窗口，从而允许你自己来处理这些通知。因此通知消息进入一个窗口，而子窗口实际上却位于另一个窗口——工具栏上。 GADGETS 采取下列步骤来完成这一点：

```
hwndCombo = CreateWindow ("combobox",
                          NULL,
                          WS_CHILD | WS_VISIBLE |
                          CBS_DROPDOWN,
                          x, y, cx, cy,
                          hwndParent,
                          (HMENU) 100,
                          hInst,
                          0) ;

// Set toolbar as combobox window parent
SetParent (hwndCombo, hwndTB) ;
```

12.2.2.10 缩放有子窗口的工具栏

基于工具栏的子窗口必须保证工具栏足够高以容纳该子窗口，从而引起另一些复杂情况。但是，对于以前的 Windows 外壳——如 Windows NT 3.51 等——可能是一个问题，而在新的 Windows 95 外壳中却不是。由于你的某些用户使用的可能正是 Windows NT 的这一版本，因此你可能应该在两种环境中对你的代码进行测试。

最简单的解决方案是缩放子窗口自身，这将适用于除了组合框之外所有类型的控制。组合框要复杂一些，因为它只会因为要容纳其当前字体才会对自己进行缩放。因此，你至少可以间接地通过设置字体来控制一个组合框的大小。你可以通过创建一个正确大小的字体然后以 WM_SETFONT “减肥丸”消息的形式将字体提供给组合框，从而达到对组合框的控制。只要一个控制比工具栏按钮短，那么 TB_AUTOSIZE 消息将处理所有调整工具栏以适应其父窗口大小变化的繁琐工作（参见前面对移动和缩放工具栏的讨论）。

当无法强迫子窗口缩小时，你就只能放大工具栏。 GADGETS 演示了这种方法，同时这也是下面讨论的基础。

为了获取对一个工具栏大小和位置的完全控制，创建带有 CCS_NORESIZE 风格标志的工具栏。除了该风格标志的名字所揭示的含义外，它实际上是要防止工具栏自己缩放自己和自己移动自己。如果设置了这个标志，那么 TB_AUTOSIZE 消息不会得到任何处理，因此移动和缩放工具栏完全由你来负责。当父窗口接收到一条 WM_SIZE 消息时，你必须计算出窗口的位置(x, y)和大小(cx, cy)，然后通过调用一个诸如 MoveWindow 之类的例程来进行改变。

对于其父窗口顶部只有一个工具栏的应用系统来说，确定位置是容易的：(0, 0)。确定宽度也是直接的，因为 WM_SIZE 在 wParam 的低位字中提供父窗口的宽度。为了响应 WM_SIZE 消息而对工具栏进行缩放，需要进行如下的处理：

```
case WM_SIZE :
     {
     int cx = LOWORD (lParam) ;
     MoveWindow (hwndToolBar, 0, 0, cx, cyToolBar, TRUE) ;
     [other program lines]
     return 0 ;
     }
```

缺少的是工具栏的高度（ cyToolBar ）。 GADGETS 的解决方案涉及两个查询：一个是对组合框大小的查询，另一个是对工具栏项目加上一个因数 5 （这是为工具栏硬编码的边空值）的查询。工具栏大小是这两个值中最大的一个：

```
// 计算工具栏高度
GetWindowRect (hwndCombo, &r) ;
cyToolBar = r.bottom - r.top +1;
cyToolBar += y;
cyToolBar += (2 * GetSystemMetrics (SM_CYBORDER)) ;
ToolBar_GetItemRect (hwndTB, 0, &r) ;
cyToolBar = max (cyToolBar, r.bottom + 5) ;
```

12.2.2.11　子窗口工具提示支持

除了处理工具提示的 TTN_NEEDTEXT 通知外，使工具提示对工具栏按钮可用是不可能的。对工具栏上的一个子窗口，这还需要另外一个步骤：在工具提示控制维护的工具列表中增加子窗口。

在一个工具提示的工具列表中增加新的工具，这项工作涉及到发送给工具提示一个 TTM_ADDTOOL 消息。但是，首先需要工具栏的工具提示的句柄，该句柄在工具栏发送 TB_GETTOOLTIP 消息时由工具栏提供：

```
hwnd = ToolBar_GetToolTip (hwndToolBar) ;
```

TTM_ADDTOOL 消息要求一个指向 TOOLINFO 结构的指针，该结构描述了工具提示控制的一个工具。 TOOLINFO 在 COMMCTRL.H 中定义如下：

```
typedef struct tagTOOLINFOA
     {
```

```
UINT cbSize ;
UINT uFlags ;
HWND hwnd ;
UINT uId ;
RECT rect ;
HINSTANCE hinst ;
LPSTR lpszText ;
} TOOLINFOA, NEAR *PTOOLINFOA, FAR *LPTOOLINFOA ;
```

将 TOOLINFO 的大小赋给 cbSize 成员，为工具提示控制提供特定于控制的版本信息，忽略填充这个值将导致工具提示控制忽略 TTM_ADDTOOL 消息。

uFlags 域允许对工具定义的一些细化工作。当将一个子窗口句柄作为工具传送时，需要一个标志： TTF_IDISHWND ，它表示项目标识符域 uId 是一个窗口句柄。另一个标志 TTF_CENTERTIP 导致工具提示将它自己在一个工具下居中显示，而不采用缺省方式，即在鼠标光标的右下方显示。第三个标志 TTF_SUBCLASS 要求工具提示控制将工具窗口子类化，来监测鼠标消息。如果没有这个标志，那么你必须使用一些其它的技术来将鼠标消息的详细信息传递给工具提示控制。

另一个域 hwnd 标识包含工具的窗口，在这里当然是工具栏。

uId 域是一个工具创建者（如工具栏）赋给每个工具的一个任意的标识符。例如，一个工具栏为其中的每个按钮赋予一个从 0 开始计数的索引，并且，当在工具提示的列表中增加每个按钮时，工具栏将这些索引传递给工具提示。当这个标识符中包含一个窗口句柄时，比如当整个子窗口构成一个工具时， uFlags 域中的 TTF_IDISHHWND 标志会将这一事实通知工具提示。

工具在其父窗口中的位置由 rect 来标识，它允许工具提示控制对工具栏窗口中的单个按钮进行选中测试。当整个窗口是一个工具——即，当 TTF_IDISHWND 标志被设置时——这个域将被忽略。

为工具显示的实际文本是通过 hinst 和 lpszText 域（标识一个字符串资源）的结合来获得的。另一种方法是通过将 lpszText 设置为 LPSTR_TEXTCALLBACK ，来请求一个 TTN_NEEDTEXT 通知。

下面所示是 GADGETS 如何为其工具栏中的组合框增加工具提示支持的：

```
TOOLINFO ti ;
ZeroMemory (&ti, sizeof (TOOLINFO)) ;
ti.cbSize = sizeof (TOOLINFO) ;
ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS ;
ti.hwnd   = hwndToolBar ;
ti.uId    = (UINT) (HWND) hwndComboBox ;
ti.lpszText = LPSTR_TEXTCALLBACK ;
bSuccess = ToolTip_AddTool (hwndTT, &ti) ;
```

关于组合框，值得提到的一点是它们是两个或者多个控制的组合：组合框容器、一个编辑或者静态控制和一个列表框。为了提供完整的工具提示支持，可见的部件应该加入工具提示的工具列表之中。 GADGETS 为编辑控制增加了工具提示支持，如下所示：

```
// 为组合框的编辑控制增加工具提示
hwndEdit = GetWindow (hwndComboBox, GW_CHILD) ;
ti.cbSize = sizeof (TOOLINFO) ;
ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS ;
ti.hwnd   = hwndToolBar ;
ti.uId    = (UINT) (HWND) hwndEdit ;
ti.lpszText = LPSTR_TEXTCALLBACK ;
bSuccess = ToolTip_AddTool (hwndTT, &ti) ;
```

一旦工具提示知道了有关子窗口的信息，那么当鼠标在该子窗口上稍停留一会，就将出现工具提示。关于这种监测有一点神奇的地方，这是通过带有 TTF_SUBCLASS 标志而请求的子类化过程实现的。这个子类化过程负责监测鼠标消息——并且，当检测到一个鼠标消息时，它向工具提示发送一条 TTM_RELAYEVNET 消息。

如果由于某些原因，你决定不使用子类化过程的技术——可能因为它会干扰已经在执行的子类化过程。你可以修改子控制的窗口过程，使得在收到一个鼠标消息时简单地发送出一个 TTM_RELAYEVENT 消息。你还可以增加一个 WH_GETMESSAGE “钩子”来监视指定窗口的鼠标消息。可能最容易的方法是在你程序的消息循环中增加一小块代码。作为鼠标（以及键盘、定时器和绘图）消息进入一个程序的主要入口，当 TTF_SUBCLASS 标志不能满足你的需要时，修改消息循环是一种好的选择。如果 GADGETS 不使用这个标志，它的消息循环应该是这样的：

```
while (GetMessage (&msg, NULL, 0, 0))
     {
     if (pMsg->hwnd == hwndCombo || pMsg->hwnd == hwndEdit)
          {
          if (pMsg->message >= WM_MOUSEFIRST &&
              pMsg->message <= WM_MOUSELAST)
               {
               ToolTip_RelayEvent (hwndTT, pMsg) ;
               }
          }

     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
     }
```

12.2.2.12　定制工具栏

工具栏在运行时可以由最终用户进行修改，做到这一点的主要技术是 Customize Toolbar（定制工具栏）对话框，如图 12-6 所示。这个对话框显示两个列表：一个是可用按钮列表，另一个是当前位于工具栏上的按钮。用户可以单击两个列表之间的 Add 和 Remove 按钮。另外，两个列表框都是拖动列表框，因此用户可以在列表中进行选择和拖动对象，或者从一个列表选择并拖动到另一个列表中。通过向工具栏发送 TB_CUSTOMIZE 消息，可以唤起这个对话框。

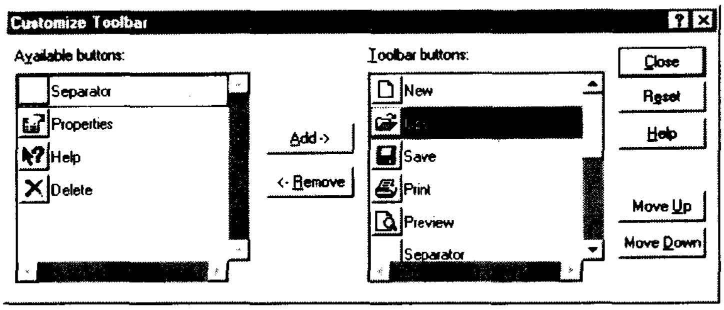


图 12-6 Customize Toolbar对话框

两个风格标志提供了通过鼠标输入来访问工具栏的功能。使用 CCS_ADJUSTABLE 标志创建的工具栏将响应在工具栏内的双击，显示定制对话框。（这个标志在激活程序控制下的定制对话框时是不必要的；在前面已经说明，只需简单地发送 TB_CUSTOMIZE 消息即可。）另外，用户可以通过使用 Shift+单击操作来选中工具栏上的按钮并将它们拖动到工具栏上的不同位置，从而对按钮进行移动。用户还可以通过选中按钮并将它们拖离工具栏，从而删除按钮。 TBSTYLE_ALTDRAG 标志将按钮拖动改成使用 Alt+单击（如在 Microsoft Word for Windows 中所采用的）而不是 Shift+单击。

工具栏定制的关键是适当地处理相关的通知消息。还在定制对话框显示之前，它就会发送三个最初的查询： TBN_QUERYINSERT 、 TBN_QUERYDELETE 和 TBN_GETBUTTONINFO 。这些相关的通知消息总结在如下所示的表中：

工具栏定制通知	描述
TBN_BEGINADJUST 和 TBN_ENDADJUST	定制对话框的生命周期由这两个通知消息开始和结束。
TBN_QUERYINSERT	询问是否可以在指定位置插入按钮。当定制对话框刚启动时，回答 TRUE ；否则，对话框将消失。
TBN_QUERYDELETE	询问是否可以删除按钮。如果可以，则返回 TRUE ，否则返回 FALSE 。
TBN_GETBUTTONINFO	一系列这样的消息可以查询能够在工具栏中显示的所有按钮的信息。那些当前不在工具栏上的按钮被放在一个可用按钮的列表中。
TBN_TOOLBARCHANGE	工具栏按钮被移动、删除或插入。
TBN_RESET	用户单击了定制对话框上的 Reset 按钮。
TBN_CUSTHELP	用户单击了定制对话框上的 Help 按钮。

TBN_QUERYINSERT 通知消息询问是否可以将任何一个新按钮插入工具栏中。如果回答是“不”（返回值为零或者 FALSE ），那么对话框将在屏幕上闪烁一下就消失，这可能会使用户很不高兴。为了保证定制对话框在屏幕上驻留一段时间，你需要对第一个这样的查询提供一个“是”（非零或者 TRUE ）的回答：

```
case WM_NOTIFY:
    {
    LPNMHDR pnmh = (LPNMHDR) lParam;

    //允许工具栏被定制
    if (pnmh->code == TBN_QUERYINSERT) ;
        {
        return 1;
        }
```

发送一系列的 TBN_QUERYDELETE 通知消息——每个工具栏按钮一个，询问各个按钮是否可以被删除。回答将导致对 Remove 压入按钮的允许和禁止：TRUE 将允许一个工具栏按钮被删除，而 FALSE 将禁止工具栏按钮被删除。

一系列的 TBN_GETBUTTONINFO 通知消息为工具栏的父窗口提供了一个机会，让它标识所有可能在工具栏上出现的按钮。对每个请求，你提供可用按钮的一个 TBBUTTON 描述，以及一个在 Customize Toolbar 对话框的列表框中描述该按钮的字符串；返回 FALSE 则表示没有更多的按钮了。这个查询的结果将填充“ Available buttons ”（可用按钮）列表框，但是只有当前位于工具栏中的按钮才会真正地出现在这个列表中。为两个按钮回答这个通知消息的一种冗长的方法如下：

```
LPTBNOTIFY ptbn = (LPTBNOTIFY) lParam ;
switch (ptbn->iItem)
    {
    case 0 :
        lstrcpy (ptbn->pszText, "Help") ;
        ptbn->tbButton.iBitmap = STD_HELP ;
        ptbn->tbButton.idCommand = 11 ;
        ptbn->tbButton.fsState = TBSTATE_ENABLED ;
        ptbn->tbButton.fsStyle = TBSTYLE_BUTTON ;
        ptbn->tbButton.dwData = 0 ;
        ptbn->tbButton.iString = 10 ;
        return 1 ;
    case 1 :
        lstrcpy (ptbn->pszText, "Delete") ;
        ptbn->tbButton.iBitmap = STD_DELETE ;
        ptbn->tbButton.idCommand = 12 ;
        ptbn->tbButton.fsState = TBSTATE_ENABLED ;
        ptbn->tbButton.fsStyle = TBSTYLE_BUTTON ;
        ptbn->tbButton.dwData = 0 ;
        ptbn->tbButton.iString = 11 ;
        return 1 ;
    default :
        return 0 ;
    }
```

由于你可能已经获得了一个 TBBUTTON 的数组，因此回答这个消息的一种更快的方法是使用 memcpy 为收到的每个通知消息拷贝一个 TBBUTTON 。下面是 GADGETS 如何使用这种方法的：

```
lstrcpy (ptbn->pszText, GetString (ptbn->iItem)) ;
memcpy (&ptbn->tbButton, &tbb[iButton], sizeof (TBBUTTON)) ;
```

12.2.3 状态栏

状态栏是仅供输出的窗口，通常位于程序主窗口的底部。状态栏最通常的用途是当用户浏览菜单项时提供一些帮助性的提示信息——就像服务员为餐馆菜单上的这个或者那个选择作一些简洁的解释一样。当菜单没被浏览时，程序通常在状态栏中显示关于程序的一些补充信息，比如换档键—— Caps Lock 、 Num Lock 和 Scroll Lock ——的状态等细节，以及一些特定于程序的信息，比如在字处理程序中，通常会显示当前的页、行或者列。

通用控制状态栏是双模式的，因此你可以处理一个状态栏的两种基本用法——菜单项描述和其它程序状态信息，而不会遇到什么冲突。在菜单描述模式（“简单模式”）中，状态栏成为一个突出的条，显示单行文本。在程序状态模式中，状态栏显示一个或多个文本小窗格，其中每一个被称为一个状态栏“部件”。单个状态栏部件可以被创建为下凹的、弹出的或者无边界的。另外，你可以在状态栏上增加子窗口（如一个时钟窗口或者一个进展条窗口）。当在简单模式和非简单模式之间进行切换时，状态栏保存一组隐藏部件，而显示一组可见部件。

12.2.3.1 创建状态栏

创建状态栏的最简单方法是调用 CreateStatusWindow：

```
hwndStatusBar = CreateStatusWindow (WS_CHILD |
                            WS_VISIBLE |
                            WS_CLIPSIBLINGS |
                            CCS_BOTTOM,
                            "Ready",
                            hwndParent,
                            2) ;
```

这个函数调用 CreateWindow ，创建一个子窗口，其父窗口为 hwndParent ，窗口文本为“ Ready ”（状态栏将在其第一个窗格中显示此文本），子窗口 ID 为 2 ，窗口风格标志用第一个参数来定义。

下面的表格总结了设计一个状态栏时可以使用的风格标志。你总是可以使用 SBARS_SIZEGRIP，因为它可以显示用户所期望的对角影线缩放标志。其余的风格标志修改工具栏最初的大小和位置，从而使工具栏可以位于非缺省 CCS_BOTTOM 的其它位置，缺省风格标志将使工具栏位于其父窗口客户区的底部。

类别	风格标志	描述
外观	SBARS_SIZEGRIP	在状态栏的右边显示对角影线缩放标志，提供“抓取这里以缩放”的功能。

续表

类别	风格标志	描述
初始位置	CCS_TOP	将状态栏放置在父窗口的顶部
	CCS_BOTTOM	将状态栏放置在父窗口的底部（缺省）
	CCS_NOMOVEY	禁止沿 y 轴移动
禁止自动移动/自动缩放	CCS_NOPARENTALIGN	状态栏只设置它自己的高度(cy)，不设置它自己的位置(x, y)和宽度(cx)。为了进行正确的处理，在创建之后发送一条消息进行缩放。
	CCS_NORESIZE	禁止所有的自动移动和自动缩放功能特性，并禁止 CCS_TOP、CCS_BOTTOM、CCS_NOMOVEC、CS_NOPARENTALIGN，你必须显式地移动和缩放状态栏。

12.2.3.2 移动和缩放状态栏

当状态栏的父窗口改变大小时（即当父窗口收到一条 WM_SIZE 消息时），状态栏必须被移动和缩放以保证它位于父窗口的客户区底部。虽然工具栏通用控制（在本章前面讨论过）会响应 TB_AUTOSIZE 消息对自己进行缩放，但是对状态栏却不存在类似的机制。因此，你必须提供如下所示的代码：

```
case WM_SIZE :
     {
     int cxParent = LOWORD (lParam) ;
     int cyParent = HIWORD (lParam) ;
     int x, y, cx, cy ;
     RECT rWindow ;

     // Keep status window height the same
     GetWindowRect (hwndStatusBar, &rWindow) ;
     cy = rWindow.bottom - rWindow.top ;

     x = 0 ;
     y = cyParent - cy ;
     cx = cxParent ;
     MoveWindow (hwndStatusBar, x, y, cx, cy, TRUE) ;
     }
```

这段代码保持状态栏的高度不变，只修改它的宽度和位置，从而使其占据父窗口区域中的正确位置。

12.2.3.3 支持菜单浏览

用户在浏览菜单以便确定哪个菜单项提供需要的服务时，希望 Window 程序能够显示帮助性的状态栏文本。即使一个不经常使用 Windows 的用户也能很快明白一个单词的菜单

项名字所隐藏的内容要比它所揭示的多。并且即使富有经验的用户已经很熟悉标准菜单命令的角色了，他们仍然会努力地试图去理解程序特定菜单项的意义。

当用户浏览菜单中的菜单项时，窗口的菜单会发送一条 WM_MENUSELECT 消息，并象我们在第 10 章中讨论过的，当真正选择了一个特定的菜单项时，还将发送一条 WM_COMMAND 消息。为了支持帮助性菜单提示的显示，你将响应这两个消息中的第一个，即 WM_MENUSELECT 消息。

为了简化处理这个消息以及在状态栏上显示文本的过程，通用控制库提供 MenuHelp 例程。这个例程假定菜单帮助文本存在于一个字符串表和一个数据结构中，从而通过菜单项可以获得字符串表的 ID 。 MenuHelp 例程定义如下：

```
void MenuHelp (UINT      uMsg,           //WM_MENUSELECT
               WPARAM    wParam,         //wParam 参数
               LPARAM    lParam,         //lParam 参数
               HMENU     hMainMenu,      //主菜单句柄
               HINSTANCE hInst,          //实例句柄
               HWND      hwndStatus,     //状态栏句柄
               UINT      *lpwIDs) ;      //字符串表数组
```

第一个参数 uMsg 必须是 WM_MENUSELECT ，虽然这个参数的出现（以及在 Win32 文档中的说明）表明了在设计这个例程时，就已经考虑了对处理 WM_COMMAND 消息的帮助。wParam 和 lParam 窗口过程参数作为第二和第三个参数。这三个参数一起描述了用户正在浏览菜单系统的哪个部分以及浏览的是一个菜单项，还是一个弹出式菜单，或者是一个系统菜单。

第六个参数 hwndStatus 是状态栏窗口的句柄。 MenuHelp 函数发送一个特定于控制的消息 SB_SIMPLE ，将状态栏设置为单窗格（简单）模式并显示适当的文本。然后，当用户停止浏览时， MenuHelp 发送另一个 SB_SIMPLE 消息，将状态栏返回到多窗格（非简单）模式。

MenuHelp 使用其它三个参数—— hMainMenu 、 hInst 和 lpwIDs ——来确定浏览不同的菜单项时，相应地显示哪个字符串。 hInst 参数标识包含菜单字符串（你可能希望使用一个 LoadString 调用装入该字符串）的字符串表所在的模块，它可以是一个 DLL 的实例句柄，或者是当前运行程序的实例句柄。

使 MenuHelp 正确工作的技巧是发送第四和第七个参数： hMainMenu 和 lpwIDs 的正确值。设置这些值并保证其正确是一件要求技巧的工作，因为有三类可供考虑的选项：菜单命令、弹出菜单和系统菜单。另一个复杂之处是 Win32 文档建议 lpwIDs 是可以由 MenuHelp 搜索的数组。虽然使用数组来存储字符串资源的基数值是可以理解的，但是你必须自己分析这个数组，因为 MenuHelp 函数不会为你这样做。

12.2.3.4 浏览菜单项

MenuHelp 函数通过将命令 ID 值（从 wParam 的低位字中获得）加上 lpwIDs 包含的值，从而计算出菜单项的资源字符串 ID 。例如，下面对 MenuHelp 的调用在状态栏上显示第 125 号字符串：

```
UINT uiStringBase = 100;
WPARAM wParam = 25;
MenuHelp (WM_MENESELECT, wParam, lParam, NULL,
        hInst, hwndStatus, &uiStringBase) ;
```

如这个例子所示，对于有关菜单命令项的显示，菜单句柄值 hMainMenu 可以是 NULL，因为 MenuHelp 在它的计算中不使用这个值。

将菜单命令项与资源字符串进行匹配的最简单的方法是给这二者赋予同样的值，从而允许你将字符串的基数值设置为 0 并忽略之。

12.2.3.5 浏览弹出式菜单项

对于弹出式菜单项来说，MenuHelp 函数通过将弹出式菜单的索引（从 0 开始计数）加上 lpwIDs 参数包含的值来计算出状态栏资源字符串。为了能够正确工作，MenuHelp 的第四个参数 hMainMenu 必须是弹出式菜单的父窗口的句柄。如下所示的结构提供了一种便捷的方式将菜单句柄映射为一个资源字符串的基数：

```
typedef struct tagPOPUPSTRING
    {
    HMENU hMenu;
    UINT   uiString;
    } POPUPSTRING;
```

在 GADGETS 程序中，有三个菜单包含弹出式菜单，这个数据结构的定义如下：

```
POPSTRING popstr[5];
```

该结构在创建状态栏时被初始化，如下所示：

```
HMENU hMainMenu = GetMenu (hwndParent) ;
popstr[0].hMenu = hMainMenu;
popstr[0].uiString = IDS_MAIN_MENU;
popstr[1].hMenu = GetSubMenu (hMainMenu, 2) ;
popstr[1].uiString = IDS_TOOLBAR_MENU;
popstr[2].hMenu = GetSubMenu (hMainMenu,3) ;
popstr[2].uiString = IDS_STATUSBAR_MENU;
popstr[3].hMenu = NULL;
popstr[3].uiString = 0;
```

当收到一条 WM_MENUSELECT 消息时，lParam 参数中包含父菜单句柄。让 MenuHelp 获取正确的字符串资源涉及到对数组本身的搜索以及将地址作为最后一个参数传给 MenuHelp，下面显示了 GADGETS 是如何完成这项工作的：

```
if ((fuFlags & MF_POPUP) &&
    (!(fuFlags & MF_SYSMENU)))
     {
     for (iMenu = 1 ; iMenu < MAX_MENUS ; iMenu++)
```

```
        {
        if ((HMENU) lParam == popstr[iMenu].hMenu)
            {
            hMainMenu = (HMENU) lParam ;
            break ;
            }
        }
    }
```

为了使这段代码正确地工作， hMainMenu 必须被设置为弹出式菜单的父菜单的句柄。当我们只感兴趣处理我们自己的弹出式菜单时，明确地排除了对系统菜单的处理。

12.2.3.6 浏览系统菜单

MenuHelp 函数为系统菜单和系统菜单项提供帮助性的状态栏文本，所要求的是来自 WM_MENUSELECT 消息中的 wParam 和 lParam 参数，这和其它类型的菜单项传递这些参数一样。而且， hMainMenu 的值必须是实际的系统菜单句柄，除此而外，也可以使用 NULL 。

12.2.3.7 例子

将对菜单项、弹出式菜单和系统菜单的所有处理综合在一起，就是下面给出的 GADGETS 如何处理 WM_MENUSELECT 消息，以便正确地显示帮助性的状态栏文本：

```
LRESULT
Statusbar_MenuSelect (HWND hwnd, WPARAM wParam, LPARAM lParam)
    {
    UINT fuFlags = (UINT) HIWORD (wParam) ;
    HMENU hMainMenu = NULL ;
    int iMenu = 0 ;

    // Handle non-system popup menu descriptions
    if ((fuFlags & MF_POPUP) &&
        (!(fuFlags & MF_SYSMENU)))
        {
        for (iMenu = 1 ; iMenu < MAX_MENUS ; iMenu++)
            {
            if ((HMENU) lParam == popstr[iMenu].hMenu)
                {
                hMainMenu = (HMENU) lParam ;
                break ;
                }
            }
        }

    // Display helpful text in status bar
    MenuHelp (WM_MENUSELECT, wParam, lParam, hMainMenu, hInst,
              hwndStatusBar, &((UINT) popstr[iMenu].hMenu)) ;

    return 0 ;
```

```
}
```

12.2.4　GADGETS 程序

GADGETS 示例程序将所讨论的三个控制综合在一起：工具栏、工具提示和状态栏。如图 12-7 所示，GADGETS 有一个带组合框的工具栏和一个带对角影线缩放标志的状态栏。它还有一个客户区，其中包含了一个列表框，在框中显示为这三种控制中的每一个接收的所有通知码。为了帮助对工具栏和状态栏的不同风格的理解，GADGETS 让你能够允许和禁止这些风格标志并显示每个风格标志的直接效果。图 12-8 中为 GADGETS 的源代码。

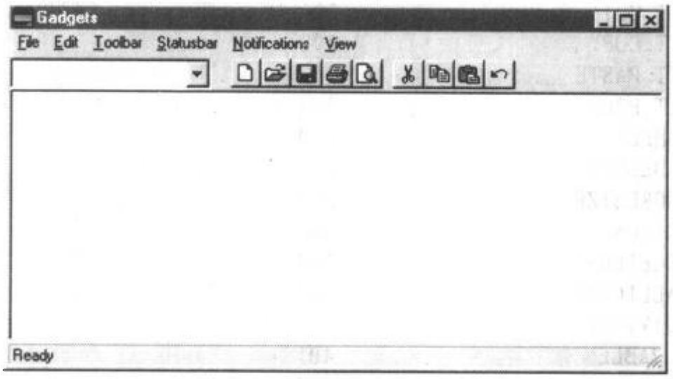


图 12-7　GADGETS的显示

GADGETS.MAK

```
#-----------------------
# GADGETS.MAK make file
#-----------------------

gadgets.exe : gadgets.obj notifdef.obj statbar.obj \
              toolbar.obj tooltip.obj gadgets.res
     $(LINKER) $(GUIFLAGS) -OUT:gadgets.exe gadgets.obj \
     notifdef.obj statbar.obj toolbar.obj tooltip.obj \
     gadgets.res $(GUILIBS)

gadgets.obj : gadgets.c comcthlp.h gadgets.h
     $(CC) $(CFLAGS) gadgets.c

notifdef.obj : notifdef.c notifdef.h
     $(CC) $(CFLAGS) notifdef.c

statbar.obj : statbar.c comcthlp.h gadgets.h
     $(CC) $(CFLAGS) statbar.c

toolbar.obj : toolbar.c comcthlp.h gadgets.h notifdef.h
     $(CC) $(CFLAGS) toolbar.c

tooltip.obj : tooltip.c comcthlp.h gadgets.h notifdef.h
     $(CC) $(CFLAGS) tooltip.c
```

```
gadgets.res : gadgets.rc gadgets.ico
     $(RC) $(RCVARS) gadgets.rc
```

GADGETS.H

```
// Resource definitions
#define IDM_FILE_NEW                100     // -- Menu commands --
#define IDM_FILE_OPEN               101
#define IDM_FILE_SAVE               102
#define IDM_FILE_SAVEAS             103
#define IDM_FILE_PRINT              104
#define IDM_FILE_PREVIEW            105
#define IDM_FILE_EXIT               106
#define IDM_EDIT_UNDO               200
#define IDM_EDIT_CUT                201
#define IDM_EDIT_COPY               202
#define IDM_EDIT_PASTE              203
#define IDM_EDIT_PROP               204
#define IDM_TB_HELP                 250
#define IDM_TB_DELETE               251
#define IDM_IGNORESIZE              300
#define IDM_STRINGS                 301
#define IDM_LARGEICONS              302
#define IDM_SMALLICONS              303
#define IDM_NODIVIDER               400
#define IDM_WRAPABLE                401
#define IDM_TOP                     402
#define IDM_BOTTOM                  403
#define IDM_NOMOVEY                 404
#define IDM_NOPARENTALIGN           405
#define IDM_NORESIZE                406
#define IDM_ADJUSTABLE              407
#define IDM_ALTDRAG                 408
#define IDM_TOOLTIPS                409
#define IDM_TB_CHECK                500
#define IDM_TB_ENABLE               501
#define IDM_TB_HIDE                 502
#define IDM_TB_INDETERMINATE        503
#define IDM_TB_PRESS                504
#define IDM_TB_BUTTONCOUNT          505
#define IDM_TB_GETROWS              506
#define IDM_TB_CUSTOMIZE            507
#define IDM_STAT_IGNORESIZE         600
#define IDM_STAT_SIZEGRIP           700
#define IDM_STAT_TOP                701
#define IDM_STAT_BOTTOM             702
#define IDM_STAT_NOMOVEY            703
#define IDM_STAT_NOPARENTALIGN      704
#define IDM_STAT_NORESIZE           705
#define IDM_ST_GETBORDERS           800
#define IDM_ST_GETPARTS             801
#define IDM_ST_SETTEXT              802
#define IDM_ST_SIMPLE               803
#define IDM_NOTIFICATIONS_CLEAR     900
#define IDM_VIEW_TOOLBAR            1000
#define IDM_VIEW_STATUS             1001
#define IDM_VIEW_NOTIFICATIONS      1002
```

```
#define IDM_COMBOBOX              4000
#define IDI_APP                   101      // -- Icons --
#define IDS_MAIN_MENU             71       // -- Strings --
#define IDS_MAIN_MENU1            72
#define IDS_MAIN_MENU2            73
#define IDS_MAIN_MENU3            74
#define IDS_MAIN_MENU4            75
#define IDS_MAIN_MENU5            76
#define IDS_TOOLBAR_MENU          80
#define IDS_TOOLBAR_MENU1         81
#define IDS_TOOLBAR_MENU2         82
#define IDS_TOOLBAR_MENU3         83
#define IDS_STATUSBAR_MENU        90
#define IDS_STATUSBAR_MENU1       91
#define IDS_STATUSBAR_MENU2       92
#define IDS_STATUSBAR_MENU3       93

#define IDC_TB_COMBOBOX           2000  // -- Toolbar combo box

// Toolbar functions
HWND InitToolBar (HWND hwndParent) ;
HWND RebuildToolBar (HWND hwndParent, WORD wFlag) ;
void ToolBarMessage (HWND hwndTB, WORD wMsg) ;
LRESULT ToolBarNotify (HWND hwnd, WPARAM wParam, LPARAM lParam) ;

// Tooltip functions
BOOL InitToolTip (HWND hwndToolBar, HWND hwndComboBox) ;
BOOL RelayToolTipMessage (LPMSG pMsg) ;
void CopyToolTipText (LPTOOLTIPTEXT lpttt) ;

// Status bar functions
HWND InitStatusBar (HWND hwndParent) ;
HWND RebuildStatusBar (HWND hwndParent, WORD wFlag) ;
void StatusBarMessage (HWND hwndSB, WORD wMsg) ;
LRESULT Statusbar_MenuSelect (HWND, WPARAM, LPARAM) ;

// Notification window functions
HWND ViewNotificationsToggle (HWND hwnd) ;
void DisplayNotificationDetails (WPARAM wParam, LPARAM lParam) ;
void ClearNotificationList () ;
```

NOTIFDEF.H

```
typedef struct tagCONTROLNOTIFICATIONS
     {
     UINT  nCode ;
     LPSTR pName ;
     } CONTROLNOTIFICATIONS ;

BOOL QueryNotifyText (UINT nNotifyCode, LPSTR *pName) ;
void DisplayText (LPSTR pText) ;
```

GADGETS.C

```
/*---------------------------------------
   GADGETS.C -- Gadgets for a frame window
                (c) Paul Yao, 1996
  ---------------------------------------*/
```

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ClientWndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[]  = "Gadgets" ;
BOOL bIgnoreSize = FALSE ;
HINSTANCE hInst ;
HWND hwndClient = NULL ;
HWND hwndToolBar = NULL ;
HWND hwndStatusBar = NULL ;
HWND hwndNotify = NULL ;

extern DWORD dwToolBarStyles ;
extern BOOL bStrings ;
extern BOOL bLargeIcons ;
extern BOOL bComboBox ;
extern DWORD dwStatusBarStyles ;
extern int cyToolBar ;

//-------------------------------------------------------------------
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wc ;

     hInst = hInstance ;

     wc.cbSize        = sizeof (wc) ;
     wc.lpszClassName = szAppName ;
     wc.hInstance     = hInstance ;
     wc.lpfnWndProc   = WndProc ;
     wc.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wc.hIcon         = LoadIcon (hInst, MAKEINTRESOURCE (IDI_APP)) ;
     wc.lpszMenuName  = "MAIN" ;
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wc.style         = 0 ;
     wc.cbClsExtra    = 0 ;
     wc.cbWndExtra    = 0 ;
     wc.hIconSm       = LoadIcon (hInst, MAKEINTRESOURCE (IDI_APP)) ;

     RegisterClassEx (&wc) ;

     wc.lpszClassName = "ClientWndProc" ;
     wc.hInstance     = hInstance ;
     wc.lpfnWndProc   = ClientWndProc ;
     wc.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wc.lpszMenuName  = NULL ;
```

```
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wc.style         = 0 ;
     wc.cbClsExtra    = 0 ;
     wc.cbWndExtra    = 0 ;
     wc.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wc) ;

     hwnd = CreateWindowEx (0L,
                            szAppName, szAppName,
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, cmdShow) ;
     UpdateWindow (hwnd) ;

     InitCommonControls () ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

//-------------------------------------------------------------------
void MenuCheckMark (HMENU hmenu, int id, BOOL bCheck)
     {
     int iState ;
     iState = (bCheck) ? MF_CHECKED : MF_UNCHECKED ;
     CheckMenuItem (hmenu, id, iState) ;
     }

//-------------------------------------------------------------------
LRESULT CALLBACK
WndProc (HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (mMsg)
          {
          case WM_CREATE :
               {
               // Create toolbar (source resides in toolbar.c)
               hwndToolBar = InitToolBar (hwnd) ;

               // Create status bar (source resides in statbar.c)
               hwndStatusBar = InitStatusBar (hwnd) ;

               // Create client window (contains notify list)
               hwndClient = CreateWindowEx (WS_EX_CLIENTEDGE,
                                 "ClientWndProc", NULL,
                                 WS_CHILD | WS_VISIBLE, 0, 0, 0, 0,
                                 hwnd, (HMENU) 4, hInst, NULL) ;

               return 0 ;
```

```
            }

        case WM_COMMAND :
             {
             // Toolbar button commands
             if (LOWORD (wParam) < 300)
                  {
                  char ach[80] ;
                  wsprintf (ach, "Got Command (%d)", wParam) ;
                  MessageBox (hwnd, ach, szAppName, MB_OK) ;
                  break ;
                  }

             // Menu item commands
             switch (LOWORD (wParam))
                  {
                  // Toolbar settings
                  case IDM_STRINGS :
                  case IDM_LARGEICONS :
                  case IDM_SMALLICONS :
                  case IDM_NODIVIDER :
                  case IDM_WRAPABLE :
                  case IDM_TOP :
                  case IDM_BOTTOM :
                  case IDM_NOMOVEY :
                  case IDM_NOPARENTALIGN :
                  case IDM_NORESIZE :
                  case IDM_ADJUSTABLE :
                  case IDM_ALTDRAG :
                  case IDM_TOOLTIPS :
                  case IDM_COMBOBOX :
                       DestroyWindow (hwndToolBar) ;
                       hwndToolBar = RebuildToolBar (hwnd,
                                                     LOWORD (wParam)) ;
                       break ;

                  // Toolbar messages
                  case IDM_TB_CHECK :
                  case IDM_TB_ENABLE :
                  case IDM_TB_HIDE :
                  case IDM_TB_INDETERMINATE :
                  case IDM_TB_PRESS :
                  case IDM_TB_BUTTONCOUNT :
                  case IDM_TB_GETROWS :
                  case IDM_TB_CUSTOMIZE :
                       ToolBarMessage (hwndToolBar, LOWORD (wParam)) ;
                       break ;

                  // Status bar settings
                  case IDM_STAT_SIZEGRIP :
                  case IDM_STAT_TOP :
                  case IDM_STAT_BOTTOM :
                  case IDM_STAT_NOMOVEY :
                  case IDM_STAT_NOPARENTALIGN :
                  case IDM_STAT_NORESIZE :
                       DestroyWindow (hwndStatusBar) ;
```

```
          hwndStatusBar =
               RebuildStatusBar (hwnd, LOWORD (wParam)) ;
          break ;

     // Status bar messages
     case IDM_ST_GETBORDERS :
     case IDM_ST_GETPARTS :
     case IDM_ST_SETTEXT :
     case IDM_ST_SIMPLE :
          StatusBarMessage (hwndStatusBar, LOWORD (wParam)) ;
          break ;

     // Toggle display of toolbar
     case IDM_VIEW_TOOLBAR :
          {
          RECT r ;

          if (hwndToolBar && IsWindowVisible (hwndToolBar))
               {
               ShowWindow (hwndToolBar, SW_HIDE) ;
               }
          else
               {
               ShowWindow (hwndToolBar, SW_SHOW) ;
               }

          // Resize other windows
          GetClientRect (hwnd, &r) ;
          PostMessage (hwnd, WM_SIZE, 0,
                       MAKELPARAM (r.right, r.bottom)) ;

          break ;
          }

     // Toggle display of status bar
     case IDM_VIEW_STATUS :
          {
          RECT r ;

          if (hwndStatusBar &&
              IsWindowVisible (hwndStatusBar))
               {
               ShowWindow (hwndStatusBar, SW_HIDE) ;
               }
          else
               {
               ShowWindow (hwndStatusBar, SW_SHOW) ;
               }

          // Resize other windows
          GetClientRect (hwnd, &r) ;
          PostMessage (hwnd, WM_SIZE, 0,
                       MAKELPARAM (r.right, r.bottom)) ;

          break ;
          }
```

```
            // Toggle display of notifications window
            case IDM_VIEW_NOTIFICATIONS :
                 hwndNotify = ViewNotificationsToggle (hwndClient) ;
                 break ;

            // Toggle ignore WM_SIZE to show auto-size/auto-move
            case IDM_IGNORESIZE :
            case IDM_STAT_IGNORESIZE :
                 {
                 RECT r ;

                 bIgnoreSize = !bIgnoreSize ;
                 if (bIgnoreSize)
                      {
                      ShowWindow (hwndClient, SW_HIDE) ;
                      }
                 else
                      {
                      ShowWindow (hwndClient, SW_SHOW) ;
                      GetClientRect (hwnd, &r) ;
                      PostMessage (hwnd, WM_SIZE, 0,
                                   MAKELPARAM (r.right, r.bottom)) ;
                      }
                 break ;
                 }

            // Clear contents of notification window
            case IDM_NOTIFICATIONS_CLEAR :
                 ClearNotificationList () ;
                 break ;
            }

       return 0 ;
       }

  case WM_INITMENU :
       {
       BOOL bCheck ;
       HMENU hmenu = (HMENU) wParam ;

       MenuCheckMark (hmenu, IDM_IGNORESIZE, bIgnoreSize) ;
       MenuCheckMark (hmenu, IDM_STAT_IGNORESIZE, bIgnoreSize) ;

       // Toolbar menu items
       MenuCheckMark (hmenu, IDM_STRINGS, bStrings) ;
       MenuCheckMark (hmenu, IDM_LARGEICONS, bLargeIcons) ;
       MenuCheckMark (hmenu, IDM_SMALLICONS, !bLargeIcons) ;
       MenuCheckMark (hmenu, IDM_COMBOBOX, bComboBox) ;

       bCheck = (dwToolBarStyles & CCS_NODIVIDER) ;
       MenuCheckMark (hmenu, IDM_NODIVIDER, bCheck) ;

       bCheck = (dwToolBarStyles & TBSTYLE_WRAPABLE) ;
       MenuCheckMark (hmenu, IDM_WRAPABLE, bCheck) ;
```

```
          bCheck = ((dwToolBarStyles & 3) == CCS_TOP) ;
          MenuCheckMark (hmenu, IDM_TOP, bCheck) ;

          bCheck = ((dwToolBarStyles & 3) == CCS_BOTTOM) ;
          MenuCheckMark (hmenu, IDM_BOTTOM, bCheck) ;

          bCheck = ((dwToolBarStyles & 3) == CCS_NOMOVEY) ;
          MenuCheckMark (hmenu, IDM_NOMOVEY, bCheck) ;

          bCheck = (dwToolBarStyles & CCS_NOPARENTALIGN) ;
          MenuCheckMark (hmenu, IDM_NOPARENTALIGN, bCheck) ;

          bCheck = (dwToolBarStyles & CCS_NORESIZE) ;
          MenuCheckMark (hmenu, IDM_NORESIZE, bCheck) ;

          bCheck = (dwToolBarStyles & CCS_ADJUSTABLE) ;
          MenuCheckMark (hmenu, IDM_ADJUSTABLE, bCheck) ;

          bCheck = (dwToolBarStyles & TBSTYLE_ALTDRAG) ;
          MenuCheckMark (hmenu, IDM_ALTDRAG, bCheck) ;

          bCheck = (dwToolBarStyles & TBSTYLE_TOOLTIPS) ;
          MenuCheckMark (hmenu, IDM_TOOLTIPS, bCheck) ;

          // Status bar menu items
          bCheck = (dwStatusBarStyles & SBARS_SIZEGRIP) ;
          MenuCheckMark (hmenu, IDM_STAT_SIZEGRIP, bCheck) ;

          bCheck = ((dwStatusBarStyles & 3) == CCS_TOP) ;
          MenuCheckMark (hmenu, IDM_STAT_TOP, bCheck) ;

          bCheck = ((dwStatusBarStyles & 3) == CCS_BOTTOM) ;
          MenuCheckMark (hmenu, IDM_STAT_BOTTOM, bCheck) ;

          bCheck = ((dwStatusBarStyles & 3) == CCS_NOMOVEY) ;
          MenuCheckMark (hmenu, IDM_STAT_NOMOVEY, bCheck) ;

          bCheck = (dwStatusBarStyles & CCS_NOPARENTALIGN) ;
          MenuCheckMark (hmenu, IDM_STAT_NOPARENTALIGN, bCheck) ;

          bCheck = (dwStatusBarStyles & CCS_NORESIZE) ;
          MenuCheckMark (hmenu, IDM_STAT_NORESIZE, bCheck) ;

          // View menu items
          bCheck = IsWindowVisible (hwndToolBar) ;
          MenuCheckMark (hmenu, IDM_VIEW_TOOLBAR, bCheck) ;

          bCheck = IsWindowVisible (hwndStatusBar) ;
          MenuCheckMark (hmenu, IDM_VIEW_STATUS, bCheck) ;

          bCheck = (hwndNotify != NULL) ;
          MenuCheckMark (hmenu, IDM_VIEW_NOTIFICATIONS, bCheck) ;
          return 0 ;
          }

     case WM_MENUSELECT :
```

```
          return Statusbar_MenuSelect (hwnd, wParam, lParam) ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;

     case WM_NOTIFY :
          {
          LPNMHDR pnmh = (LPNMHDR) lParam ;
          int idCtrl = (int) wParam ;

          // Display notification details in notify window
          DisplayNotificationDetails (wParam, lParam) ;

          // Toolbar notifications
          if ((pnmh->code >= TBN_LAST) &&
              (pnmh->code <= TBN_FIRST))
               {
               return ToolBarNotify (hwnd, wParam, lParam) ;
               }

          // Fetch tooltip text
          if (pnmh->code == TTN_NEEDTEXT)
               {
               LPTOOLTIPTEXT lpttt = (LPTOOLTIPTEXT) lParam ;
               CopyToolTipText (lpttt) ;
               }

          return 0 ;
          }

     case WM_SIZE :
          {
          int cx = LOWORD (lParam) ;
          int cy = HIWORD (lParam) ;
          int cyStatus ;
          int cyTB ;
          int x, y ;
          DWORD dwStyle ;
          RECT rWindow ;

          // Ignore size message to allow auto-move and auto-size
          //   features to be more clearly seen
          if (bIgnoreSize)
               return 0 ;

          // Adjust toolbar size
          if (IsWindowVisible (hwndToolBar))
               {
               dwStyle = GetWindowLong (hwndToolBar, GWL_STYLE) ;
               if (dwStyle & CCS_NORESIZE)
                    {
                    MoveWindow (hwndToolBar,
                                0, 0, cx, cyToolBar, FALSE) ;
                    }
               else
```

```
                  {
                  ToolBar_AutoSize (hwndToolBar) ;
                  }
             InvalidateRect (hwndToolBar, NULL, TRUE) ;
             GetWindowRect (hwndToolBar, &rWindow) ;
             ScreenToClient (hwnd, (LPPOINT) &rWindow.left) ;
             ScreenToClient (hwnd, (LPPOINT) &rWindow.right) ;
             cyTB = rWindow.bottom - rWindow.top ;
             }
        else
             {
             cyTB = 0 ;
             }

        // Adjust status bar size
        if (IsWindowVisible (hwndStatusBar))
             {
             GetWindowRect (hwndStatusBar, &rWindow) ;
             cyStatus = rWindow.bottom - rWindow.top ;
             MoveWindow (hwndStatusBar, 0, cy - cyStatus,
                                        cx, cyStatus, TRUE) ;
             }
        else
             {
             cyStatus = 0 ;
             }
        // Adjust client window size
        x = 0 ;
        y = cyTB ;
        cy = cy - (cyStatus + cyTB) ;
        MoveWindow (hwndClient, x, y, cx, cy, TRUE) ;
        return 0 ;
        }

   default :
        return (DefWindowProc (hwnd, mMsg, wParam, lParam)) ;
   }
}

//-----------------------------------------------------------------
LRESULT CALLBACK
ClientWndProc (HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
     {
     static COLORREF crBack ;
     static HBRUSH hbr ;

     switch (mMsg)
          {
          case WM_CREATE :
               hwndNotify = ViewNotificationsToggle (hwnd) ;
               crBack = GetSysColor (COLOR_APPWORKSPACE) ;
               hbr = CreateSolidBrush (crBack) ;
               return 0 ;

          case WM_DESTROY :
               DeleteObject (hbr) ;
```

```
                return 0 ;

           case WM_CTLCOLORLISTBOX :
                {
                DefWindowProc (hwnd, mMsg, wParam, lParam) ;
                SetBkColor ((HDC) wParam, crBack) ;
                SetBkMode ((HDC) wParam, TRANSPARENT) ;
                return (LRESULT) (HBRUSH) hbr ;
                }

           case WM_SIZE :
                {
                HWND hwndNotify = GetWindow (hwnd, GW_CHILD) ;
                int cx = LOWORD (lParam) ;
                int cy = HIWORD (lParam) ;

                // Ignore if notification window is absent
                if (hwndNotify != NULL)
                     {
                     MoveWindow (hwndNotify, 0, 0, cx, cy, TRUE) ;
                     }

                return 0 ;
                }

           default :
                return (DefWindowProc (hwnd, mMsg, wParam, lParam)) ;
           }
}
```

TOOLBAR.C

```
/*----------------------------------------
   TOOLBAR.C -- Toolbar helper functions
                (c) Paul Yao, 1996
  ----------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"
#include "notifdef.h"

HWND hwndCombo ;
HWND hwndEdit ;
HWND hwndToolTip ;
HWND hwndTB ;

int cyToolBar ;

BOOL bComboBox   = FALSE ;
BOOL bStrings    = FALSE ;
BOOL bLargeIcons = FALSE ;

DWORD dwToolBarStyles = WS_CHILD | WS_VISIBLE | WS_CLIPSIBLINGS |
                        CCS_TOP | CCS_NODIVIDER | TBSTYLE_TOOLTIPS ;

extern HINSTANCE hInst ;
```

```
char szTbStrings[]  = "New\0Open\0Save\0Print\0Preview\0"
                      "Cut\0Copy\0Paste\0Undo\0Properties\0"
                      "Help\0Delete\0" ;

TBBUTTON tbb[] =
     {
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,
     STD_FILENEW, IDM_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 0,
     STD_FILEOPEN, IDM_FILE_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 1,
     STD_FILESAVE, IDM_FILE_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 2,
     STD_PRINT, IDM_FILE_PRINT, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 3,
     STD_PRINTPRE, IDM_FILE_PREVIEW, TBSTATE_ENABLED, TBSTYLE_CHECK,
        0, 0, 0, 4,

     0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0,

     STD_CUT, IDM_EDIT_CUT, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP,
        0, 0, 0, 5,
     STD_COPY, IDM_EDIT_COPY, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP,
        0, 0, 0, 6,
     STD_PASTE, IDM_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_CHECKGROUP,
        0, 0, 0, 7,
     STD_UNDO, IDM_EDIT_UNDO, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 8,
     STD_PROPERTIES, IDM_EDIT_PROP, TBSTATE_ENABLED, TBSTYLE_CHECK,
        0, 0, 0, 9,
     STD_HELP, IDM_TB_HELP, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 10,
     STD_DELETE, IDM_TB_DELETE, TBSTATE_ENABLED, TBSTYLE_BUTTON,
        0, 0, 0, 11,
     } ;
```

```
int nCust[] = { 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, -1} ;

TBADDBITMAP tbStdLarge[] =
     {
     HINST_COMMCTRL, IDB_STD_LARGE_COLOR,
     } ;

TBADDBITMAP tbStdSmall[] =
     {
     HINST_COMMCTRL, IDB_STD_SMALL_COLOR,
     } ;

//-------------------------------------------------------------------
LPSTR GetString (int iString)
     {
     int i, cb ;
     LPSTR pString ;

     // Cycle through to requested string
     pString = szTbStrings ;
     for (i = 0 ; i < iString ; i++)
          {
          cb = lstrlen (pString) ;
          pString += (cb + 1) ;
          }

     return pString ;
     }

//-------------------------------------------------------------------
LRESULT ToolBarNotify (HWND hwnd, WPARAM wParam, LPARAM lParam)
     {
     LPNMHDR pnmh = (LPNMHDR) lParam ;
     int idCtrl = (int) wParam ;

     // Allow toolbar to be customized
     if ((pnmh->code == TBN_QUERYDELETE) ||
         (pnmh->code == TBN_QUERYINSERT))
          {
          return 1 ; // We always say "yes"
          }

     // Provide details of allowable toolbar buttons
     if (pnmh->code == TBN_GETBUTTONINFO)
          {
          LPTBNOTIFY ptbn = (LPTBNOTIFY) lParam ;
          int iButton = nCust[ptbn->iItem] ;

          if (iButton != -1)
               {
               lstrcpy (ptbn->pszText, GetString (ptbn->iItem)) ;
               memcpy (&ptbn->tbButton, &tbb[iButton], sizeof (TBBUTTON)) ;
               return 1 ;
               }
          }
```

```
     return 0 ;
     }

//------------------------------------------------------------------
HWND InitToolBar (HWND hwndParent)
     {
     int  iNumButtons ;
     LPTBBUTTON ptbb ;

     if (bComboBox)
          {
          ptbb = &tbb[0] ;
          iNumButtons = 31 ;
          }
     else
          {
          ptbb = &tbb[21] ;
          iNumButtons = 10 ;
          }

     UINT uiBitmap = (bLargeIcons) ? IDB_STD_LARGE_COLOR :
                                     IDB_STD_SMALL_COLOR ;
     hwndTB = CreateToolbarEx (hwndParent,
                    dwToolBarStyles,
                    1, 15,
                    HINST_COMMCTRL,
                    uiBitmap,
                    ptbb,
                    iNumButtons,
                    0, 0, 0, 0,
                    sizeof (TBBUTTON)) ;

     // If requested, add to string list
     if (bStrings)
          ToolBar_AddString (hwndTB, 0, szTbStrings) ;

     // Store handle to tooltip control
     hwndToolTip = ToolBar_GetToolTips (hwndTB) ;

     // Insert combo box into toolbar
     if (bComboBox)
          {
          RECT r ;
          int x, y, cx, cy ;

          // Calculate coordinates for combo box
          ToolBar_GetItemRect (hwndTB, 0, &r) ;
          x = r.left ;
          y = r.top ;
          cy = 100 ;
          ToolBar_GetItemRect (hwndTB, 18, &r) ;
          cx = r.right - x + 1 ;

          hwndCombo = CreateWindow ("combobox",
                           NULL,
```

```
                             WS_CHILD | WS_VISIBLE |
                             CBS_DROPDOWN,
                             x, y, cx, cy,
                             hwndParent,
                             (HMENU) IDC_TB_COMBOBOX,
                             hInst,
                             0) ;

          // Set toolbar as combo box window parent
          SetParent (hwndCombo, hwndTB) ;

          SendMessage (hwndCombo, CB_ADDSTRING, 0, (LPARAM) "One") ;
          SendMessage (hwndCombo, CB_ADDSTRING, 0, (LPARAM) "Two") ;
          SendMessage (hwndCombo, CB_ADDSTRING, 0, (LPARAM) "Three") ;

          // Calculate toolbar height
          GetWindowRect (hwndCombo, &r) ;
          cyToolBar = r.bottom - r.top + 1 ;
          cyToolBar += y ;
          cyToolBar += (2 * GetSystemMetrics (SM_CYBORDER)) ;
          ToolBar_GetItemRect (hwndTB, 0, &r) ;
          cyToolBar = max (cyToolBar, r.bottom + 5) ;

          // If toolbar has tooltips, initialize
          if (dwToolBarStyles & TBSTYLE_TOOLTIPS)
               InitToolTip (hwndTB, hwndCombo) ;
          }

     return hwndTB ;
     }

//-------------------------------------------------------------------
void static FlipStyleFlag (LPDWORD dwStyle, DWORD flag)
     {
     if (*dwStyle & flag)   // Flag on -- turn off
          {
          *dwStyle &= (~flag) ;
          }
     else                   // Flag off -- turn on
          {
          *dwStyle *= flag ;
          }
     }

//-------------------------------------------------------------------
HWND RebuildToolBar (HWND hwndParent, WORD wFlag)
     {
     HWND hwndTB ;
     RECT r ;

     switch (wFlag)
          {
          case IDM_STRINGS :
               bStrings = !bStrings ;
               break ;
```

```
          case IDM_LARGEICONS :
               bLargeIcons = TRUE ;
               break ;

          case IDM_SMALLICONS :
               bLargeIcons = FALSE ;
               break ;

          case IDM_NODIVIDER :
               FlipStyleFlag (&dwToolBarStyles, CCS_NODIVIDER) ;
               break ;

          case IDM_WRAPABLE :
               FlipStyleFlag (&dwToolBarStyles, TBSTYLE_WRAPABLE) ;
               break ;

          case IDM_TOP :
               dwToolBarStyles &= 0xFFFFFFFC ;
               dwToolBarStyles *= CCS_TOP ;
               break ;

          case IDM_BOTTOM :
               dwToolBarStyles &= 0xFFFFFFFC ;
               dwToolBarStyles *= CCS_BOTTOM ;
               break ;

          case IDM_NOMOVEY :
               dwToolBarStyles &= 0xFFFFFFFC ;
               dwToolBarStyles *= CCS_NOMOVEY ;
               break ;

          case IDM_NOPARENTALIGN :
               FlipStyleFlag (&dwToolBarStyles, CCS_NOPARENTALIGN) ;
               break ;

          case IDM_NORESIZE :
               FlipStyleFlag (&dwToolBarStyles, CCS_NORESIZE) ;
               break ;

          case IDM_ADJUSTABLE :
               FlipStyleFlag (&dwToolBarStyles, CCS_ADJUSTABLE) ;
               break ;

          case IDM_ALTDRAG :
               FlipStyleFlag (&dwToolBarStyles, TBSTYLE_ALTDRAG) ;
               break ;

          case IDM_TOOLTIPS :
               FlipStyleFlag (&dwToolBarStyles, TBSTYLE_TOOLTIPS) ;
               break ;

          case IDM_COMBOBOX :
               bComboBox = (!bComboBox) ;
          }

     hwndTB = InitToolBar (hwndParent) ;
```

```
     // Post parent a WM_SIZE message to resize children
     GetClientRect (hwndParent, &r) ;
     PostMessage (hwndParent, WM_SIZE, 0,
                  MAKELPARAM (r.right, r.bottom)) ;

     return hwndTB ;
     }


//-------------------------------------------------------------
void ToolBarMessage (HWND hwndTB, WORD wMsg)
     {
     switch (wMsg)
          {
          case IDM_TB_CHECK :
               {
               int nState = ToolBar_GetState (hwndTB, 1) ;
               BOOL bCheck = (!(nState & TBSTATE_CHECKED)) ;
               ToolBar_CheckButton (hwndTB, 1, bCheck) ;
               break ;
               }

          case IDM_TB_ENABLE :
               {
               int nState = ToolBar_GetState (hwndTB, 2) ;
               BOOL bEnabled = (!(nState & TBSTATE_ENABLED)) ;
               ToolBar_EnableButton (hwndTB, 2, bEnabled) ;
               break ;
               }

          case IDM_TB_HIDE :
               {
               int nState = ToolBar_GetState (hwndTB, 3) ;
               BOOL bShow = (!(nState & TBSTATE_HIDDEN)) ;
               ToolBar_HideButton (hwndTB, 3, bShow) ;
               break ;
               }

          case IDM_TB_INDETERMINATE :
               {
               int nState = ToolBar_GetState (hwndTB, 4) ;
               BOOL bInd = (!(nState & TBSTATE_INDETERMINATE)) ;
               ToolBar_Indeterminate (hwndTB, 4, bInd) ;
               break ;
               }

          case IDM_TB_PRESS :
               {
               int nState = ToolBar_GetState (hwndTB, 5) ;
               BOOL bPress = (!(nState & TBSTATE_PRESSED)) ;
               ToolBar_PressButton (hwndTB, 5, bPress) ;
               break ;
               }

          case IDM_TB_BUTTONCOUNT :
```

```
            {
            int nButtons = ToolBar_ButtonCount (hwndTB) ;
            char ach[80] ;
            wsprintf (ach, "Button Count = %d", nButtons) ;
            MessageBox (GetParent (hwndTB), ach,
                        "TB_BUTTONCOUNT", MB_OK) ;
            break ;
            }

        case IDM_TB_GETROWS :
            {
            int nRows = ToolBar_GetRows (hwndTB) ;
            char ach[80] ;
            wsprintf (ach, "Row Count = %d", nRows) ;
            MessageBox (GetParent (hwndTB), ach,
                        "TB_GETROWS", MB_OK) ;
            break ;
            }

        case IDM_TB_CUSTOMIZE :
            // ToolBar_Customize (hwndTB) ;
            SendMessage (hwndTB, TB_CUSTOMIZE, (LPARAM) &tbb[25], 5) ;
            break ;
        }
    }
```

TOOLTIP.C

```
/*-----------------------------------------
   TOOLTIP.C -- Tooltip helper functions
                (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "notifdef.h"
#include "gadgets.h"
extern BOOL bComboBox ;
extern char szTbStrings[] ;
extern HINSTANCE hInst ;
extern HWND hwndEdit ;
extern HWND hwndCombo ;
extern HWND hwndEdit ;
static HWND hwndTT ;

// Map toolbar button command to string index
int CommandToString[] =
     { IDM_FILE_NEW, IDM_FILE_OPEN, IDM_FILE_SAVE, IDM_FILE_PRINT,
       IDM_FILE_PREVIEW, IDM_EDIT_CUT, IDM_EDIT_COPY, IDM_EDIT_PASTE,
       IDM_EDIT_UNDO, IDM_EDIT_PROP, IDM_TB_HELP, IDM_TB_DELETE, -1
     } ;

//-------------------------------------------------------------------
BOOL InitToolTip (HWND hwndToolBar, HWND hwndComboBox)
     {
     BOOL bSuccess ;
     TOOLINFO ti ;
```

```
     // Fetch handle to tooltip control
     hwndTT = ToolBar_GetToolTips (hwndToolBar) ;
     if (hwndTT == NULL)
          return FALSE ;

     // Add tooltip for main combo box
     ZeroMemory (&ti, sizeof (TOOLINFO)) ;
     ti.cbSize = sizeof (TOOLINFO) ;
     ti.uFlags = TTF_IDISHWND | TTF_CENTERTIP | TTF_SUBCLASS ;
     ti.hwnd   = hwndToolBar ;
     ti.uId    = (UINT) (HWND) hwndComboBox ;
     ti.lpszText = LPSTR_TEXTCALLBACK ;
     bSuccess = ToolTip_AddTool (hwndTT, &ti) ;
     if (!bSuccess)
          return FALSE ;

     // Add tooltip for combo box's edit control
     hwndEdit = GetWindow (hwndComboBox, GW_CHILD) ;
     ti.uId    = (UINT) (HWND) hwndEdit ;
     bSuccess = ToolTip_AddTool (hwndTT, &ti) ;

     return bSuccess ;
     }

//-------------------------------------------------------------------
void CopyToolTipText (LPTOOLTIPTEXT lpttt)
     {
     int i ;
     int iButton = lpttt->hdr.idFrom ;
     int cb ;
     int cMax ;
     LPSTR pString ;
     LPSTR pDest = lpttt->lpszText ;

     // Check for combo box window handles
     if (lpttt->uFlags & TTF_IDISHWND)
          {
          if ((iButton == (int) hwndCombo) ||
              (iButton == (int) hwndEdit))
               {
               lstrcpy (pDest, "1-2-3 ComboBox") ;
               return ;
               }
          }

     // Map command ID to string index
     for (i = 0 ; CommandToString[i] != -1 ; i++)
          {
          if (CommandToString[i] == iButton)
               {
               iButton = i ;
               break ;
               }
          }
```

```
    // To be safe, count number of strings in text
    pString = szTbStrings ;
    cMax = 0 ;
    while (*pString != '\0')
         {
         cMax++ ;
         cb = lstrlen (pString) ;
         pString += (cb + 1) ;
         }

    // Check for valid parameter
    if (iButton > cMax)
         {
         pString = "Invalid Button Index" ;
         }
    else
         {
         // Cycle through to requested string
         pString = szTbStrings ;
         for (i = 0 ; i < iButton ; i++)
              {
              cb = lstrlen (pString) ;
              pString += (cb + 1) ;
              }
         }

    lstrcpy (pDest, pString) ;
   }
```

STATBAR.C

```
/*-----------------------------------------
   STATBAR.C -- Status bar helper functions
                (c) Paul Yao, 1996
  -----------------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "gadgets.h"

typedef struct tagPOPUPSTRING
     {
     HMENU hMenu ;
     UINT uiString ;
     } POPUPSTRING ;

#define MAX_MENUS 5

static POPUPSTRING popstr[MAX_MENUS] ;

DWORD dwStatusBarStyles = WS_CHILD | WS_VISIBLE |
                          WS_CLIPSIBLINGS | CCS_BOTTOM |
                          SBARS_SIZEGRIP ;
extern HINSTANCE hInst ;
extern HWND hwndStatusBar ;

//-------------------------------------------------------------------
```

```
HWND InitStatusBar (HWND hwndParent)

     {
     HWND hwndSB ;
     // Initialize values for WM_MENUSELECT message handling
     HMENU hMenu = GetMenu (hwndParent) ;
     HMENU hMenuTB = GetSubMenu (hMenu, 2) ;
     HMENU hMenuSB = GetSubMenu (hMenu, 3) ;
     popstr[0].hMenu    = 0 ;
     popstr[0].uiString = 0 ;
     popstr[1].hMenu    = hMenu ;
     popstr[1].uiString = IDS_MAIN_MENU ;
     popstr[2].hMenu    = hMenuTB ;
     popstr[2].uiString = IDS_TOOLBAR_MENU ;
     popstr[3].hMenu    = hMenuSB ;
     popstr[3].uiString = IDS_STATUSBAR_MENU ;
     popstr[4].hMenu    = 0 ;
     popstr[4].uiString = 0 ;

     hwndSB = CreateStatusWindow (dwStatusBarStyles,
                                  "Ready",
                                  hwndParent,
                                  2) ;
     return hwndSB ;
     }

//-------------------------------------------------------------------
void static FlipStyleFlag (LPDWORD dwStyle, DWORD flag)
     {
     if (*dwStyle & flag)  // Flag on -- turn off
          {
          *dwStyle &= (~flag) ;
          }
     else                  // Flag off -- turn on
          {
          *dwStyle *= flag ;
          }
     }
//-------------------------------------------------------------------
HWND RebuildStatusBar (HWND hwndParent, WORD wFlag)
     {
     HWND hwndSB ;
     RECT r ;

     switch (wFlag)
          {
          case IDM_STAT_SIZEGRIP :
               FlipStyleFlag (&dwStatusBarStyles, SBARS_SIZEGRIP) ;
               break ;
          case IDM_STAT_TOP :
               dwStatusBarStyles &= 0xFFFFFFFC ;
               dwStatusBarStyles *= CCS_TOP ;
               break ;

          case IDM_STAT_BOTTOM :
               dwStatusBarStyles &= 0xFFFFFFFC ;
```

```
                dwStatusBarStyles *= CCS_BOTTOM ;
                break ;

            case IDM_STAT_NOMOVEY :
                dwStatusBarStyles &= 0xFFFFFFFC ;
                dwStatusBarStyles *= CCS_NOMOVEY ;
                break ;

            case IDM_STAT_NOPARENTALIGN :
                FlipStyleFlag (&dwStatusBarStyles, CCS_NOPARENTALIGN) ;
                break ;

            case IDM_STAT_NORESIZE :
                FlipStyleFlag (&dwStatusBarStyles, CCS_NORESIZE) ;
                break ;
            }

     hwndSB = InitStatusBar (hwndParent) ;

     // Post parent a WM_SIZE message to resize children
     GetClientRect (hwndParent, &r) ;
     PostMessage (hwndParent, WM_SIZE, 0,
                  MAKELPARAM (r.right, r.bottom)) ;

     return hwndSB ;
     }


//-------------------------------------------------------------------
void StatusBarMessage (HWND hwndSB, WORD wMsg)
     {
     switch (wMsg)
          {
          case IDM_ST_GETBORDERS :
               {
               char ach[180] ;
               int aiBorders[3] ;

               Status_GetBorders (hwndSB, &aiBorders) ;
               wsprintf (ach, "Horiz Width = %d\n"
                         "Vert Width = %d\n"
                         "Separator Width = %d",
                         aiBorders[0], aiBorders[1],
                         aiBorders[2]) ;
               MessageBox (GetParent (hwndSB), ach,
                           "SB_GETBORDERS", MB_OK) ;
               break ;
               }

          case IDM_ST_GETPARTS :
               {
               char ach[80] ;
               int nParts = Status_GetParts (hwndSB, 0, 0) ;
               wsprintf (ach, "Part Count = %d", nParts) ;
               MessageBox (GetParent (hwndSB), ach,
                           "SB_GETPARTS", MB_OK) ;
```

```
                    break ;
                    }

               case IDM_ST_SETTEXT :
                    Status_SetText (hwndSB, 0, 0,
                                   "SB_SETTEXT Message Sent") ;
                    break ;

               case IDM_ST_SIMPLE :
                    {
                    static BOOL bSimple = TRUE ;
                    Status_Simple (hwndSB, bSimple) ;
                    bSimple = (!bSimple) ;
                    break ;
                    }
               }
          }

//-------------------------------------------------------------------
LRESULT
Statusbar_MenuSelect (HWND hwnd, WPARAM wParam, LPARAM lParam)
     {
     UINT fuFlags = (UINT) HIWORD (wParam) ;
     HMENU hMainMenu = NULL ;
     int iMenu = 0 ;

     // Handle non-system popup menu descriptions
     if ((fuFlags & MF_POPUP) &&
         (!(fuFlags & MF_SYSMENU)))
          {
          for (iMenu = 1 ; iMenu < MAX_MENUS ; iMenu++)
               {
               if ((HMENU) lParam == popstr[iMenu].hMenu)
                    {
                    hMainMenu = (HMENU) lParam ;
                    break ;
                    }
               }
          }

     // Display helpful text in status bar
     MenuHelp (WM_MENUSELECT, wParam, lParam, hMainMenu, hInst,
               hwndStatusBar, (UINT *) popstr) ;

     return 0 ;
     }
```

NOTIFDEF.C

```
/*--------------------------------------------------
   NOTIFDEF.C -- Support notification detail window
                 (c) Paul Yao, 1996
  --------------------------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <prsht.h>
```

```
#include "notifdef.h"

CONTROLNOTIFICATIONS cnLookupTable[] =
     {
     NM_OUTOFMEMORY,          "NM_OUTOFMEMORY",
     NM_CLICK,                "NM_CLICK",
     NM_DBLCLK,               "NM_DBLCLK",
     NM_RETURN,               "NM_RETURN",
     NM_RCLICK,               "NM_RCLICK",
     NM_RDBLCLK,              "NM_RDBLCLK",
     NM_SETFOCUS,             "NM_SETFOCUS",
     NM_KILLFOCUS,            "NM_KILLFOCUS",
     LVN_ITEMCHANGING,        "LVN_ITEMCHANGING",
     LVN_ITEMCHANGED,         "LVN_ITEMCHANGED",
     LVN_INSERTITEM,          "LVN_INSERTITEM",
     LVN_DELETEITEM,          "LVN_DELETEITEM",
     LVN_DELETEALLITEMS,      "LVN_DELETEALLITEMS",
     LVN_BEGINLABELEDITA,     "LVN_BEGINLABELEDITA",
     LVN_BEGINLABELEDITW,     "LVN_BEGINLABELEDITW",
     LVN_ENDLABELEDITA,       "LVN_ENDLABELEDITA",
     LVN_ENDLABELEDITW,       "LVN_ENDLABELEDITW",
     LVN_COLUMNCLICK,         "LVN_COLUMNCLICK",
     LVN_BEGINDRAG,           "LVN_BEGINDRAG",
     LVN_BEGINRDRAG,          "LVN_BEGINRDRAG",
     LVN_GETDISPINFOA,        "LVN_GETDISPINFOA",
     LVN_GETDISPINFOW,        "LVN_GETDISPINFOW",
     LVN_SETDISPINFOA,        "LVN_SETDISPINFOA",
     LVN_SETDISPINFOW,        "LVN_SETDISPINFOW",
     LVN_KEYDOWN,             "LVN_KEYDOWN",
     HDN_ITEMCHANGINGA,       "HDN_ITEMCHANGINGA",
     HDN_ITEMCHANGINGW,       "HDN_ITEMCHANGINGW",
     HDN_ITEMCHANGEDA,        "HDN_ITEMCHANGEDA",
     HDN_ITEMCHANGEDW,        "HDN_ITEMCHANGEDW",
     HDN_ITEMCLICKA,          "HDN_ITEMCLICKA",
     HDN_ITEMCLICKW,          "HDN_ITEMCLICKW",
     HDN_ITEMDBLCLICKA,       "HDN_ITEMDBLCLICKA",
     HDN_ITEMDBLCLICKW,       "HDN_ITEMDBLCLICKW",
     HDN_DIVIDERDBLCLICKA,    "HDN_DIVIDERDBLCLICKA",
     HDN_DIVIDERDBLCLICKW,    "HDN_DIVIDERDBLCLICKW",
     HDN_BEGINTRACKA,         "HDN_BEGINTRACKA",
     HDN_BEGINTRACKW,         "HDN_BEGINTRACKW",
     HDN_ENDTRACKA,           "HDN_ENDTRACKA",
     HDN_ENDTRACKW,           "HDN_ENDTRACKW",
     HDN_TRACKA,              "HDN_TRACKA",
     HDN_TRACKW,              "HDN_TRACKW",
     TVN_SELCHANGINGA,        "TVN_SELCHANGINGA",
     TVN_SELCHANGINGW,        "TVN_SELCHANGINGW",
     TVN_SELCHANGEDA,         "TVN_SELCHANGEDA",
     TVN_SELCHANGEDW,         "TVN_SELCHANGEDW",
     TVN_GETDISPINFOA,        "TVN_GETDISPINFOA",
     TVN_GETDISPINFOW,        "TVN_GETDISPINFOW",
     TVN_SETDISPINFOA,        "TVN_SETDISPINFOA",
     TVN_SETDISPINFOW,        "TVN_SETDISPINFOW",
     TVN_ITEMEXPANDINGA,      "TVN_ITEMEXPANDINGA",
     TVN_ITEMEXPANDINGW,      "TVN_ITEMEXPANDINGW",
     TVN_ITEMEXPANDEDA,       "TVN_ITEMEXPANDEDA",
```

```
        TVN_ITEMEXPANDEDW,      "TVN_ITEMEXPANDEDW",
        TVN_BEGINDRAGA,         "TVN_BEGINDRAGA",
        TVN_BEGINDRAGW,         "TVN_BEGINDRAGW",
        TVN_BEGINRDRAGA,        "TVN_BEGINRDRAGA",
        TVN_BEGINRDRAGW,        "TVN_BEGINRDRAGW",
        TVN_DELETEITEMA,        "TVN_DELETEITEMA",
        TVN_DELETEITEMW,        "TVN_DELETEITEMW",
        TVN_BEGINLABELEDITA,    "TVN_BEGINLABELEDITA",
        TVN_BEGINLABELEDITW,    "TVN_BEGINLABELEDITW",
        TVN_ENDLABELEDITA,      "TVN_ENDLABELEDITA",
        TVN_ENDLABELEDITW,      "TVN_ENDLABELEDITW",
        TVN_KEYDOWN,            "TVN_KEYDOWN",
        TTN_NEEDTEXTA,          "TTN_NEEDTEXTA",
        TTN_NEEDTEXTW,          "TTN_NEEDTEXTW",
        TTN_SHOW,               "TTN_SHOW",
        TTN_POP,                "TTN_POP",
        TCN_KEYDOWN,            "TCN_KEYDOWN",
        TCN_SELCHANGE,          "TCN_SELCHANGE",
        TCN_SELCHANGING,        "TCN_SELCHANGING",
        TBN_GETBUTTONINFOA,     "TBN_GETBUTTONINFOA",
        TBN_GETBUTTONINFOW,     "TBN_GETBUTTONINFOW",
        TBN_BEGINDRAG,          "TBN_BEGINDRAG",
        TBN_ENDDRAG,            "TBN_ENDDRAG",
        TBN_BEGINADJUST,        "TBN_BEGINADJUST",
        TBN_ENDADJUST,          "TBN_ENDADJUST",
        TBN_RESET,              "TBN_RESET",
        TBN_QUERYINSERT,        "TBN_QUERYINSERT",
        TBN_QUERYDELETE,        "TBN_QUERYDELETE",
        TBN_TOOLBARCHANGE,      "TBN_TOOLBARCHANGE",
        TBN_CUSTHELP,           "TBN_CUSTHELP",
        UDN_DELTAPOS,           "UDN_DELTAPOS",
        PSN_SETACTIVE,          "PSN_SETACTIVE",
        PSN_KILLACTIVE,         "PSN_KILLACTIVE",
        PSN_APPLY,              "PSN_APPLY",
        PSN_RESET,              "PSN_RESET",
        PSN_HELP,               "PSN_HELP",
        PSN_WIZBACK,            "PSN_WIZBACK",
        PSN_WIZNEXT,            "PSN_WIZNEXT",
        PSN_WIZFINISH,          "PSN_WIZFINISH",
        PSN_QUERYCANCEL,        "PSN_QUERYCANCEL"
        } ;

int NOTIFY_COUNT = sizeof (cnLookupTable) / sizeof (CONTROLNOTIFICATIONS) ;
static HWND hwndNotify = NULL ;

extern HINSTANCE hInst ;

//-------------------------------------------------------------------
// QueryNotifyText: Convert notification codes into text
//-------------------------------------------------------------------
BOOL QueryNotifyText (UINT nNotifyCode, LPSTR *pName)
     {
     BOOL bFound = FALSE ;
     int  iNotify ;

     for (iNotify = 0 ; iNotify < NOTIFY_COUNT ; iNotify++)
```

```
          {
          if (cnLookupTable[iNotify].nCode == nNotifyCode)
               {
               *pName = cnLookupTable[iNotify].pName ;
               return TRUE ;
               }
          }

     // Unknown notification code
     *pName = "** Unknown **" ;
     return FALSE ;
     }

//-------------------------------------------------------------------
// ViewNotificationsToggle: Display/hide notification window
//-------------------------------------------------------------------
HWND ViewNotificationsToggle (HWND hwnd)
     {
     int x, y, cx, cy ;
     RECT rWindow ;

     if (hwndNotify)
          {
          DestroyWindow (hwndNotify) ;
          hwndNotify = NULL ;
          }
     else
          {
          GetClientRect (hwnd, &rWindow) ;
          x = 0 ;
          y = 0 ;
          cx = rWindow.right  - rWindow.left ;
          cy = rWindow.bottom - rWindow.top ;
          hwndNotify = CreateWindowEx (0L,
                            "LISTBOX", NULL,
                            LBS_NOINTEGRALHEIGHT |
                            WS_CHILD   |
                            WS_VISIBLE |
                            WS_VSCROLL,
                            x, y, cx, cy,
                            hwnd, (HMENU) 1, hInst, NULL) ;
          }

     return hwndNotify ;
     }

//-------------------------------------------------------------------
void DisplayText (LPSTR pText)
     {
     int iIndex ;

     if (hwndNotify == NULL)
          return ;

     iIndex = ListBox_AddString (hwndNotify, pText) ;
     ListBox_SetTopIndex (hwndNotify, iIndex) ;
```

```
     }

//--------------------------------------------------------------------
void DisplayNotificationDetails (WPARAM wParam, LPARAM lParam)
     {
     LPNMHDR pnmh ;
     LPSTR   pName ;

     if (hwndNotify == NULL)
          return ;

     pnmh = (LPNMHDR) lParam ;
     QueryNotifyText (pnmh->code, &pName) ;
     DisplayText (pName) ;
     }

//--------------------------------------------------------------------
void ClearNotificationList ()
     {
     if (hwndNotify == NULL)
          return ;

     ListBox_ResetContent (hwndNotify) ;
     }
```

GADGETS.RC

```
#include "gadgets.h"
#include <windows.h>

IDI_APP ICON    DISCARDABLE     "gadgets.ico"

MAIN MENU DISCARDABLE
     {
     POPUP "&File"
          {
          MENUITEM "&New",                        IDM_FILE_NEW
          MENUITEM "&Open",                       IDM_FILE_OPEN
          MENUITEM "&Save",                       IDM_FILE_SAVE
          MENUITEM "Save &As...",                 IDM_FILE_SAVEAS
          MENUITEM SEPARATOR
          MENUITEM "&Print...",                   IDM_FILE_PRINT
          MENUITEM "Print Pre&view...",           IDM_FILE_PREVIEW
          MENUITEM SEPARATOR
          MENUITEM "&Exit",                       IDM_FILE_EXIT
          }
     POPUP "&Edit"
          {
          MENUITEM "&Undo\tCtrl+Z",               IDM_EDIT_UNDO
          MENUITEM SEPARATOR
          MENUITEM "&Cut\tCtrl+X",                IDM_EDIT_CUT
          MENUITEM "&Copy\tCtrl+C",               IDM_EDIT_COPY
          MENUITEM "&Paste\tCtrl+V",              IDM_EDIT_PASTE
          MENUITEM SEPARATOR
          MENUITEM "Pr&operties",                 IDM_EDIT_PROP
          }
     POPUP "&Toolbar"
```

```
          {
          POPUP "St&yles"
               {
               MENUITEM "CCS_NODIVIDER",           IDM_NODIVIDER
               MENUITEM "TBSTYLE_WRAPABLE",        IDM_WRAPABLE
               MENUITEM SEPARATOR
               MENUITEM "CCS_TOP",                 IDM_TOP
               MENUITEM "CCS_BOTTOM",              IDM_BOTTOM
               MENUITEM "CCS_NOMOVEY",             IDM_NOMOVEY
               MENUITEM SEPARATOR
               MENUITEM "CCS_NOPARENTALIGN",       IDM_NOPARENTALIGN
               MENUITEM "CCS_NORESIZE",            IDM_NORESIZE
               MENUITEM SEPARATOR
               MENUITEM "CCS_ADJUSTABLE",          IDM_ADJUSTABLE
               MENUITEM "TBSTYLE_ALTDRAG",         IDM_ALTDRAG
               MENUITEM SEPARATOR
               MENUITEM "TBSTYLE_TOOLTIPS",        IDM_TOOLTIPS
               }
          MENUITEM "&Ignore WM_SIZE",         IDM_IGNORESIZE
          MENUITEM SEPARATOR
          POPUP "&Messages"
               {
               MENUITEM "TB_CHECKBUTTON",          IDM_TB_CHECK
               MENUITEM "TB_ENABLEBUTTON",         IDM_TB_ENABLE
               MENUITEM "TB_HIDEBUTTON",           IDM_TB_HIDE
               MENUITEM "TB_INDETERMINATE",        IDM_TB_INDETERMINATE
               MENUITEM "TB_PRESSBUTTON",          IDM_TB_PRESS
               MENUITEM SEPARATOR
               MENUITEM "TB_BUTTONCOUNT",          IDM_TB_BUTTONCOUNT
               MENUITEM "TB_GETROWS",              IDM_TB_GETROWS
               MENUITEM SEPARATOR
               MENUITEM "TB_CUSTOMIZE",            IDM_TB_CUSTOMIZE
               }
          MENUITEM SEPARATOR
          MENUITEM "&Large Icons",            IDM_LARGEICONS
          MENUITEM "&Small Icons",            IDM_SMALLICONS
          MENUITEM SEPARATOR
          MENUITEM "S&trings",                IDM_STRINGS
          MENUITEM "&Combo Box",              IDM_COMBOBOX
          }
     POPUP "&Statusbar"
          {
          POPUP "&Styles"
               {
               MENUITEM "SBARS_SIZEGRIP",          IDM_STAT_SIZEGRIP
               MENUITEM SEPARATOR
               MENUITEM "CCS_TOP",                 IDM_STAT_TOP
               MENUITEM "CCS_BOTTOM",              IDM_STAT_BOTTOM
               MENUITEM "CCS_NOMOVEY",             IDM_STAT_NOMOVEY
               MENUITEM SEPARATOR
               MENUITEM "CCS_NOPARENTALIGN",       IDM_STAT_NOPARENTALIGN
               MENUITEM "CCS_NORESIZE",            IDM_STAT_NORESIZE
               }
          MENUITEM "&Ignore WM_SIZE",         IDM_STAT_IGNORESIZE
          MENUITEM SEPARATOR
          POPUP "&Messages"
```

```
                {
                MENUITEM "SB_GETBORDERS",             IDM_ST_GETBORDERS
                MENUITEM "SB_GETPARTS",               IDM_ST_GETPARTS
                MENUITEM "SB_SETTEXT",                IDM_ST_SETTEXT
                MENUITEM SEPARATOR
                MENUITEM "SB_SIMPLE",                 IDM_ST_SIMPLE
                }
            }
        POPUP "&Notifications"
            {
            MENUITEM "&Clear",                    IDM_NOTIFICATIONS_CLEAR
            }
        POPUP "&View"
            {
            MENUITEM "&Toolbar",                  IDM_VIEW_TOOLBAR
            MENUITEM "&Status Bar",               IDM_VIEW_STATUS
            MENUITEM "&Notifications",            IDM_VIEW_NOTIFICATIONS
            }
        }

    STRINGTABLE DISCARDABLE
        {
        IDS_MAIN_MENU
            "Create, open, save, print documents or quit program"
        IDS_MAIN_MENU1          "Undo, cut, copy, paste, and properties"
        IDS_MAIN_MENU2          "Toolbar styles, messages, and creation options"
        IDS_MAIN_MENU3          "Status bar styles and messages"
        IDS_MAIN_MENU4          "Clear notifications window"
        IDS_MAIN_MENU5
            "Show or hide toolbar, status bar, and notifications window"
        IDS_TOOLBAR_MENU        "Set toolbar styles and re-create toolbar"
        IDS_TOOLBAR_MENU1       "placeholder"
        IDS_TOOLBAR_MENU2       "placeholder"
        IDS_TOOLBAR_MENU3       "Send messages to toolbar"
        IDS_STATUSBAR_MENU      "Set status bar styles and re-create status bar"
        IDS_STATUSBAR_MENU1     "placeholder"
        IDS_STATUSBAR_MENU2     "placeholder"
        IDS_STATUSBAR_MENU3     "Send messages to status bar"
        IDM_FILE_NEW            "Creates a new document"
        IDM_FILE_OPEN           "Open an existing document"
        IDM_FILE_SAVE           "Save the active document"
        IDM_FILE_SAVEAS         "Save the active document with a new name"
        IDM_FILE_PRINT          "Prints the active document"
        IDM_FILE_PREVIEW        "Displays full pages as they will be printed"
        IDM_FILE_EXIT           "Quits program"
        IDM_EDIT_UNDO           "Reverse the last action"
        IDM_EDIT_CUT
            "Cuts the selection and puts it on the Clipboard"
        IDM_EDIT_COPY
            "Copies the selection and puts it on the Clipboard"
        IDM_EDIT_PASTE
            "Inserts the Clipboard contents at the insertion point"
        IDM_EDIT_PROP
            "Opens property sheet for currently selected item"
        IDM_IGNORESIZE
            "Toggle WM_SIZE handling to show autosize/auto move"
```

```
IDM_STRINGS            "Creates toolbar with strings"
IDM_LARGEICONS         "Creates toolbar with large icons"
IDM_SMALLICONS         "Creates toolbar with small icons"
IDM_COMBOBOX           "Creates toolbar with combobox"
IDM_NODIVIDER          "Toggle divider above toolbar"
IDM_WRAPABLE           "Toggle toolbar resizing for narrow window"
IDM_TOP                "Toggle placing toolbar at top of parent"
IDM_BOTTOM             "Toggle placing toolbar at bottom of parent"
IDM_NOMOVEY            "Toggle inhibit moving window on Y-axis"
IDM_NOPARENTALIGN      "Toggle inhibit aligning to parent"
IDM_NORESIZE           "Toggle inhibit any sizing or moving"
IDM_ADJUSTABLE         "Toggle ability for user to customize toolbar"
IDM_ALTDRAG            "Toggle Alt-click to drag buttons"
IDM_TOOLTIPS           "Toggle tooltip support"
IDM_TB_CHECK           "Toggle button 0 checked state"
IDM_TB_ENABLE          "Toggle button 1 enabled state"
IDM_TB_HIDE            "Toggle button 2 hidden state"
IDM_TB_INDETERMINATE   "Toggle button 3 indeterminate state"
IDM_TB_PRESS           "Toggle button 4 pressed state"
IDM_TB_BUTTONCOUNT     "Query button count"
IDM_TB_GETROWS         "Query row count"
IDM_TB_CUSTOMIZE       "Request customize dialog box"
IDM_STAT_IGNORESIZE
     "Toggle WM_SIZE handling to show autosize/auto move"
IDM_STAT_SIZEGRIP      "Status bar to have sizing grip"
IDM_STAT_TOP           "Toggle placing status bar at top of parent"
IDM_STAT_BOTTOM        "Toggle placing status bar at bottom of parent"
IDM_STAT_NOMOVEY       "Toggle inhibit moving window on Y-axis"
IDM_STAT_NOPARENTALIGN "Toggle inhibit aligning to parent"
IDM_STAT_NORESIZE      "Toggle inhibit any sizing or moving"
IDM_ST_GETBORDERS      "Query size of status bar borders"
IDM_ST_GETPARTS        "Query number of status bar parts"
IDM_ST_SETTEXT         "Set text in status bar"
IDM_ST_SIMPLE          "Toggle status bar simple state"
IDM_NOTIFICATIONS_CLEAR "Clear contents of notification window"
IDM_VIEW_TOOLBAR       "Show/hide toolbar"
IDM_VIEW_STATUS        "Show/hide status bar"
IDM_VIEW_NOTIFICATIONS "Show/hide notification window"
}
```

GADGETS.ICO

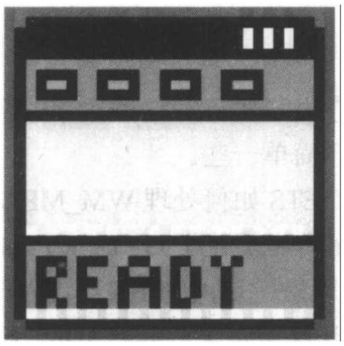


图 12-8　GADGETS的源文件

当其主窗口接收到一条 WM_CREATE 消息时，GADGETS 创建一个工具栏、一个状

态栏和一个客户区。创建时，每个窗口的位置没有关系，因为在 WM_CREATE 之后总是跟着一条 WM_SIZE 消息，那时 GADGETS 将重新定位它的子窗口。

GADGETS 的客户窗口包含一个列表框，其中将显示由主窗口收到的通知(WM_NOTIF)消息的细节，这些消息由三个通用控制——工具栏、状态栏和工具提示控制——在遇到一个值得通知的事件时发送出来。工具栏发送鼠标按钮的通知消息和工具栏定制查询；状态栏也发送有关查询的通知消息；当鼠标光标暂停在工具栏的某个工具上时，工具提示控制要求获得在工具提示窗口中显示的文本。

前两个弹出式菜单 File 和 Edit 实际上是模拟工具栏按钮动作的假菜单，其它的四个菜单 Toolbar 、 Statusbar 、 Notificatioin 和 View 控制 GADGETS 将演示的控制特性。实际操作一下 GADGETS ，就可以看到工具栏、状态栏和工具提示的主要特性。

12.2.4.1 控制GADGETS的工具栏

GADGETS 让你看到影响工具栏（和其它控制）的三个因素之间的关系，这三个因素是：窗口风格、特定于控制的消息和通知消息。通过选择 Toolbar 菜单中的 Style 子菜单，可以设置窗口的风格；通过选择 Toolbar 菜单中的 Messages 子菜单，可以发送特定于控制的消息，并观察显示在客户区大小的列表框中的通知消息。你可以通过在 Toolbar 菜单上出现的其它选项来控制各种工具栏特征——图标大小、字符串的外观和一个组合框的外观。

启动时， GADGETS 的工具栏有小的图标和工具提示支持，没有字符串和组合框，它位于其父窗口的顶部，但是可以通过设置各种工具栏风格来改变自动缩放和自动移动特性。当你在练习工具栏的定位时，可能希望通过选择适当的菜单项来禁止 GADGETS 对工具栏的定位。

GADGETS 给它的工具栏发送一组示例工具栏消息，其中大多数是针对单个按钮的，如 TB_CHECKBUTTON 是选择按钮和取消选择的， TB_ENABLEBUTTON 是启用和禁止按钮的。还有两个查询消息使你可以询问按钮的个数和行的数目。

TB_CUSTOMIZE 消息激活 Customize Toolbar 对话框，该对话框让用户增加或者删除工具栏中的项。正确操纵定制对话框的关键是为工具栏通知消息 TBN_QUERYINSERT 、 TBN_QUERYDELETE 和 TBN_GETBUTTONINFO 提供正确的应答。当你定制 GADGETS 的工具栏时，注意观察通知消息流，从而了解这些请求的顺序和流程。

12.2.4.2 控制GADGETS的状态栏

GADGETS 有一个菜单用于控制状态栏风格并向它发送消息。这个菜单的命令集要比工具栏菜单少得多，因为状态在某种程度上要比工具栏简单一些。

GADGETS 状态栏中最令人感兴趣的方面是 GADGETS 如何处理 WM_MENUSELECT 消息并在状态窗口中显示帮助性文本。所有的状态栏字符串都位于 GADGETS 的字符串表中。为了简化菜单项命令 ID 到字符串资源 ID 之间的映射，这两个 ID 设置为相同的值。例如， File Open 菜单命令的命令 ID 由 IDM_FILE_OPEN 所定义，同时 IDM_FILE_OPEN 也是字符串“ Open an existing document ”的字符串资源 ID 。

为了在选择一个弹出式菜单时提供帮助性文本， GADGETS 将菜单句柄映射为字符串

ID 。GADGETS 有十个弹出式菜单，分别由三个父菜单所拥有：主菜单条、工具栏弹出式菜单和状态栏菜单。在三个父菜单中的弹出式菜单的字符串资源状态栏描述以字符串 IDS_MAIN_MENU 、 IDS_TOOLBAR_MENU 和 IDS_STATUSBAR_MENU 开始。GADGETS 将父菜单句柄映射到这三个值，并将这些值传递给 MenuHelp 函数，从而显示正确的字符串文本。

12.3 属性表格

“ Property sheets ”（属性表格）——有时被称为“ tabbed dialog ”（ Tab 停留位对话框)——提供在复合对话框中组合几种对话框的能力。在属性表格中，单个的对话框被称为——“属性页”。图 12-9 显示了可以在 Windows 95 桌面上使用的 Display Properties 属性表格。（要访问这个属性表格，可以通过单击鼠标右键来激活外壳环境菜单并选择“ Properties ”菜单项。）在这个属性表格中的各种页允许用户精细地调整桌面选项，如墙纸、屏幕保护程序、颜色和字体，还可以修改显示驱动设置来改变可使用的颜色、设备分辨率，甚至可以改变当前的显示驱动程序。

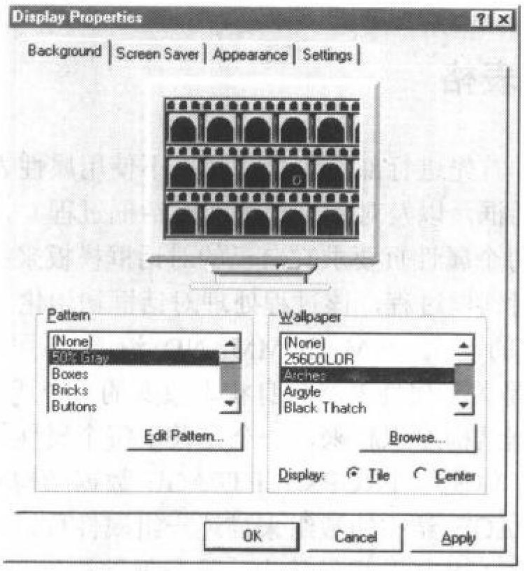


图 12-9 Display Properties属性表格

属性表格提供了修改一个复杂对象（在此例中为屏幕显示）的所有设置的能力，它通过一组方便地连接在一起的属性页来实现其功能。每页有它自己的标志，用户可以单击这个标志来调出指定的页。还有一个可选的 Apply 按钮允许用户在不离开属性表格的情况下预览所作的修改。如果结果不令人满意，那么可以再进行另一种修改——可能回到原来的设置——而同时属性表格仍然是可用的。

另一类和属性表格相关的复合对话框是向导(wizard)。一个向导是一组对话框，它们接收一种特定顺序的用户输入来完成特定的任务。图 12-10 显示了 Windows 95 中的 Add Printer Wizard 的前六页，它通过一系列的对话框允许用户为安装一个新的系统打印机输入

所需要的所有信息。向导的编程界面和属性表格的编程接口相同，虽然它们的用户界面有点不同：OK 和 Cancel 按钮由 Back 和 Next 按钮所取代。这些按钮提供了在向导页之间漫游的唯一方法（而不是在属性表格中使用的 Tab 键）。在最后一页上，向导显示一个 Finish 按钮，当按下这个按钮时，向导将消失，并开始执行请求向导完成的任务。

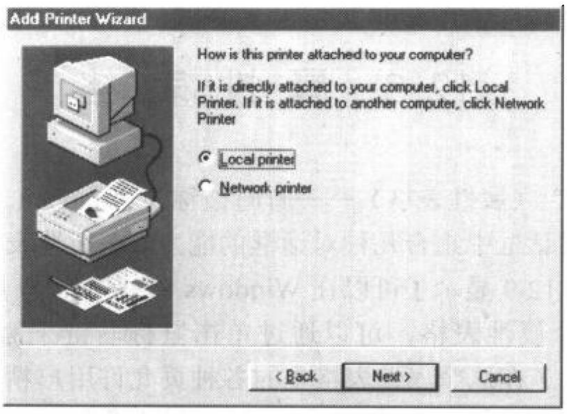


图 12-10　Add Printer Wizard中的一页

12.3.1　创建属性表格

要创建属性表格，你首先进行部件收集（如果不使用属性表格，你需要用这些部件来创建一组相互独立的对话框，以及对话框模板和对话框过程），然后再填充一些数据结构并调用 PropertySheet 。每个属性页要求它自己的对话框模板来描述单个控制的外观。每个属性页还需要它自己的对话框过程，该过程处理对话框初始化（ WM_INITDIALOG ）消息，并处理来自控制的通知(WM_COMMAND)消息。属性页对话框过程还处理 WM_NOTIFY ，此消息是关于属性表格自身状态改变的一个通知消息。

这些部件由两个数据结构组织起来，一个是关于每个属性页的，还有一个是描述属性表格整体属性的头结构。你使用 PROPSHEETPAGE 数据结构来定义每个属性页的属性，并用一个 PROPSHEETPAGE 结构的数组来描述一组属性页。一个指向属性页结构的指针被包含在 PROPSHEETHEADER 数据结构中，该数据结构包含属性表格的属性。

12.3.1.1　属性页对话框模板

为属性页使用的对话框模板几乎和你创建普通对话框时所使用的一样。属性页对话框模板包含常规的预定义窗口控制，如编辑、列表框和压入按钮；它还包含通用控制，如动画、拖动列表、跟踪条和上下控制。当使用常规对话框模板时，一个属性页模板中的每个控制都有一个唯一的控制 ID ，在运行时，你以它为参数调用 GetDlgItem 以获得一个窗口句柄。

普通对话框模板和属性页模板之间的区别是属性页模板通常不包含 OK 和 Cancel 这样的压入按钮。属性表格创建并维护一组由所有的属性页共享的压入按钮。对一个常规的属

性表格来说，缺省的按钮是 OK 、 Cancel 和 Apply ，不过，如果 Apply 按钮不能适应你的需要，那么你也可以设置一个标志来禁止 Apply 按钮。你还可以设置一个标志来请求属性表格显示一个 Help 按钮，该按钮将在支持帮助的属性页被启用。向导使用一组稍微不同的按钮： Back 和 Next 在向导的页之间切换，并且在最后一页上使用一个 Finish 按钮。

另一个关于对话框模板的问题涉及单个属性页对话框的大小和形状。在一个特定的属性表格中，你会希望其中的所有页都具有相同的大小和形状，因为属性表格会调节它自己的大小来容纳最宽和最高的属性页。精细地调整对话框模板的外观总是一件既费时又枯燥的工作；在属性表格环境中，还有更多的工作是去协调不同对话框模板的外观。

12.3.1.2 PROPSHEETPAGE结构

在你为每个属性页创建了对话框模板并且至少有了对话框过程的轮廓之后，创建属性表格中的下一步就是为每个属性页填充一个 PROPSHEETPAGE 结构。 PROPSHEETPAGE 在 PRSHT.H 中定义如下：

```
typedef struct _PROPSHEETPAGE
     {
     DWORD           dwSize ;          // = sizeof (PROPSHEETPAGE)
     DWORD           dwFlags ;
     HINSTANCE       hInstance ;
     union
          {
          LPCSTR          pszTemplate ;  // Default
          LPCDLGTEMPLATE pResource ;     // PSP_DLGINDIRECT
          } DUMMYUNIONNAME ;
     union
          {
          HICON       hIcon ;          // PSP_USEHICON
          LPCSTR      pszIcon ;        // PSP_USEICONID
          } DUMMYUNIONNAME2 ;
     LPCSTR          pszTitle ;        // PSP_USETITLE
     DLGPROC         pfnDlgProc ;
     LPARAM          lParam ;
     LPFNPSPCALLBACKA pfnCallback ;    // PSP_USECALLBACK
     UINT FAR *pcRefParent ;           // PSP_USEREFPARENT
     } PROPSHEETPAGE, FAR *LPPROPSHEETPAGE ;
```

这个列表中的注释为每个成员提供了解释。注释中的 PSP_标志表示该标志必须被包含在 dwFlags 域中以便激活一个指定的数据结构成员。下面是有关 PROPSHEETPAGE 中各个成员含义的详细信息：

- dwSize——标志属性页的当前版本；应该被设置为 sizeof(PROPSHEETPAGE)。
- dwFlags——包含通过 C 按位或操作符组合起来的一个或多个 PSP_标志。大多数标志控制 PROPSHEETPAGE 结构中的成员是否可以使用或者可以被忽略。还有两个不是用于这个目的的标志，它们是 PSP_HASHELP 和 PSP_RTLREADING 。PSP_HASHELP 标志使得激活一个特定页时启用 Help 按钮（ PSH_HASHELP 标志必须在 Help 按钮要出现的 PROPSHEETHEADER 中出现）。PSP_RTLREADING 使属性页使用从右到左的阅读（用于希伯莱语和阿拉伯语）。

- hInstance——标识资源，包括对话框模板（pszTemplate）和图标（pszIcon），从哪些可执行文件装入。
- 对话框模板由资源中装入的模板 pszTemplate （缺省）标识，或者由内存中装入的模板 pResource （要求 dwFlags 中的 PSP_DLGINDIRECT ）标识。
- 可以在属性页的标志中显示一个图标，该图标可以由图标句柄 hIcon（要求 dwFlags 中的 PSP_USEHICON ）标识，也可以由图标资源 ID ， pszIcon （要求 dwFlags 中的 PSP_USEINCONID ）标识。
- 属性页标志的缺省标题是在对话框模板中指定的标题，除非在 dwFlags 中设置了 PSP_USETITLE ，后一种情况下将使用 pzsTitle 。
- pfnDlgProc——指定属性页对话框过程。
- lParam——指定传递给属性页的回调函数的一个初始化值，由 pfnCallback 参数标识。
- pfnCallback 是一个回调函数（要求 dwFlags 中的 PSP_USECALLBACK ），在创建属性页之前以及清除属性页之后将调用这个函数。
- pcRefParent 标识属性表格的引用记数。它是一个指向 UINT 的指针，当属性页被创建时该数加 1 ，当属性页被清除时该数减 1 。在任何一个时刻，由这个指针指向的值准确地告诉你存在多少属性页。

本章的属性表格示例程序 PROPERTY 按如下所示填充其 PROPSHEETPAGE 结构：

```
PROPSHEETPAGE   pspage[2] ;
// Zero out property page data structure
ZeroMemory (&pspage, 2 * sizeof (PROPSHEETPAGE)) ;

// Init window style page data
pspage[0].dwSize      = sizeof (PROPSHEETPAGE) ;
pspage[0].dwFlags     = PSP_USECALLBACK | PSP_USEICONID ;
pspage[0].hInstance   = hInst ;
pspage[0].pszTemplate = MAKEINTRESOURCE (IDD_STYLES) ;
pspage[0].pszIcon     = MAKEINTRESOURCE (IDI_PAGE1) ;
pspage[0].pfnDlgProc  = StyleDlgProc ;
pspage[0].lParam      = (LPARAM) &dwChildStyle ;
pspage[0].pfnCallback = StylePageProc ;

// Init extended window style page data
pspage[1].dwSize      = sizeof (PROPSHEETPAGE) ;
pspage[1].dwFlags     = PSP_USECALLBACK | PSP_USEICONID |
                        PSP_HASHELP ;
pspage[1].hInstance   = hInst ;
pspage[1].pszTemplate = MAKEINTRESOURCE (IDD_EXSTYLES) ;
pspage[1].pszIcon     = MAKEINTRESOURCE (IDI_PAGE2) ;
pspage[1].pfnDlgProc  = ExStyleDlgProc ;
pspage[1].lParam      = (LPARAM) &dwChildExStyle ;
pspage[1].pfnCallback = ExStylePageProc ;
```

PROPSHEETPAGE 的 pfnDlgProc 成员是一个对话框过程的指针。除了大多数对话框过程要处理的 WM_INITDIALOG 和 WM_COMMAND 消息之外，属性页对话框过程必须处理第三条消息， WM_NOTIFY 。下面是属性页对话框过程的大致框架：

```
BOOL CALLBACK
EmptyPageDlgProc (HWND hwndDlg, UINT msg,
                  WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               [dialog initialization]
               break ;

          case WM_COMMAND :
               [notifications from regular dialog controls]
               break ;

          case WM_NOTIFY :
               {
               LPNMHDR pnmh = (LPNMHDR) lParam ;
               [property sheet notifications]
               }
          }
     }
```

PROPSHEETPAGE 的 pfnCallback 成员指向一个回调函数，在创建一个属性页之前和清除一个属性页之后将调用该函数。使用这个函数的一个原因是为了能够获得 RPOPSHEETPAGE 的 lParam 成员，它包含一个由属性页的创建者发送给属性页的初始化值（如指向将被修改的数据的指针等等）。下面是属性页回调函数的大致框架：

```
static LPARAM InputData ;

UINT CALLBACK
EmptyPageProc (HWND hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
     {
     switch (uMsg)
          {
          case PSPCB_CREATE :
               InputData = ppsp->lParam ;
               return TRUE ;

          case PSPCB_RELEASE :
               return 0 ;
          }
     }
```

在当前的实现中，窗口句柄参数 hwnd 总是 NULL 。对于属性页创建之前的调用， uMsg 为 PSPCB_CREATE ；对于清除属性页之后的调用， uMsg 为 PSPCB_RELEASE 。虽然只有在属性页真正被创建时才会发送创建通知，但释放通知却总是会被发送给每个属性页（无论是否创建某个属性页）。

当填充了一个 PROPSHEETPAGE 结构数组之后，创建属性表格过程的下一步应该是引用属性表格头结构 PROPSHEETHEADER 的 ppsp 成员中的数组。

12.3.1.3 PROPSHEETHEADER结构

PROPSHEETHEADER 结构在 PRSHT.H 中定义如下：

```
typedef struct _PROPSHEETHEADER
    {
    DWORD           dwSize ;            // = sizeof (PROPSHEETHEADER)
    DWORD           dwFlags ;
    HWND            hwndParent ;
    HINSTANCE       hInstance ;
    union
         {
         HICON       hIcon ;            // PSH_USEHICON
         LPCSTR      pszIcon ;          // PSH_USEICONID
         }DUMMYUNIONNAME ;
    LPCSTR          pszCaption ;
    UINT            nPages ;
    union
         {
         UINT        nStartPage ;       // (Default)
         LPCSTR      pStartPage ;       // PSH_USEPSTARTPAGE
         }DUMMYUNIONNAME2 ;
    union
         {
         LPCPROPSHEETPAGE ppsp ;        // PSH_PROPSHEETPAGE
         HPROPSHEETPAGE FAR *phpage ;
         }DUMMYUNIONNAME3 ;
    PFNPROPSHEETCALLBACK pfnCallback ; // PSH_USECALLBACK
    } PROPSHEETHEADER, FAR *LPPROPSHEETHEADER ;
```

在这段代码中的注释提供了每个成员的注解。在注解中出现 PSH_标志表示必须在 dwFlags 域中包含相应标志来激活特定的数据结构成员。下面是 PROPSHEETHEADER 中各个成员的具体含义。

- dwSize 为版本控制标识出 PROPSHEETHEADER 的大小，它应该被设置为 sizeof (PROPSHEETHEADER)。
- dwFlags 包含使用 C 按位或操作符组合起来的一个或者多个 PSH_标志。有些标志控制 PROPSHEETHEADER 的某个成员是否如上面所说明的那样使用，其它标志的细节将在下面说明。
- hwndParent 标识属性表格的父窗口。对于模态属性表格，其父窗口被禁用。
- 对于使用资源（如图标等）的属性表格， hInstance 参数标识从哪个模块中装入资源。
- 属性表格可以选择使用图标，图标可以由图标句柄 hIcon （要求 dwFlags 中的 PSH_USEICON ），也可以由图标资源 pszIcon （要求 dwFlags 中的 PSH_USEICONID ）来标识。
- 属性表格标题由 pszCaption 字符串定义。
- 属性表格中的页数（即属性页数组的大小）为 nPages 。
- 要显示的第一属性页可以是 nStartPage ，这是一个从 0 开始计数的索引，或者也

可以是 pStartPage （要求 dwFlags 中的 PSH_USEPSTARTPAGE），这是要显示的第一页的标题。

- 一个属性表格的页使用两种类型之一的数组来定义。第一种选择是采用由属性表格页句柄组成的数组，这些句柄是由 CreatePropertySheetPage （缺省）返回并由 phpage 来标识的；第二种选择是采用一个由 PROPSHEETPAGE 结构（要求 dwFlags 中的 PSH_PROPSHEETPAGE 标志）组成的数组，这一数组由 ppsp 标识。
- 除了每个属性页的对话框过程之外，属性表格本身还有一个回调函数，在创建属性表格上的任何一页之前都会调用这个函数，这个函数由 pfnCallback 标识。

dwFlags 允许对属性表格进行精细的调整，有些标志指示是否应该使用或者忽略 PROPSHEETHEADER 的某个成员。在前面的结构定义中，单个成员使用标志的名字来表示。其它的标志总结在下表中：

标志	含义
PSH_HASHELP	在属性表格中显示 Help 按钮，该按钮将在激活一个带有 PSP_HASHELP 标志的页时被启用。按下这个按钮时，它发送一条通知码为 PSN_HELP 的 WM_NOTIFY 消息。
PSH_MODELESS	创建一个非模态属性表格。
PSH_NOAPPLYNOW	隐藏 Apply 按钮。
PSH_PROPTITLE	当标题由 PROPSHEETHEADER 的 pszCaption 成员定义之后，在标题条中添加单词"Properties"。
PSH_WIZARD	创建向导而不是属性表格

12.3.1.4　PropertySheet函数

实际创建属性表格的函数是 PropertySheet，该函数使用一个参数：指向 PROPSHEETHEADER 结构的指针。属性表格缺省是模态的，因此当用户单击 OK 时，PropertySheet 函数返回 TRUE，当用户单击 Cancel 时，函数返回 FALSE。对 PropertySheet 的调用看起来像一个典型的 DialogBox 调用：

```
if (PropertySheet (&pshead)) ;
    {
    [用户按下 OK --接受修改]
    }
else
    {
    [用户按下 Cancel --放弃修改]
    }
```

如果要创建一个非模态属性表格，你可以当调用 PropertySheet 创建属性表格时，在属性表格头结构中设置 PSH_MODELESS 标志。这种情况下，PropertySheet 函数返回的不是一个 Boolean 值，而是一个属性表格窗口的句柄：

```
// 创建非模态属性表格--返回 HWND
```

```
HWND hwndPropSheet = (HWND) ;PropertySheet (&pshead) ;
```

12.3.2 属性表格对话框过程

每个属性页有它自己的对话框过程。在创建属性页时，对话框过程接收一条 WM_INITDIALOG 消息，作为对这个消息的响应，对话框过程初始化页中的控制。不过，属性表格采用一种偷懒的方法来创建单个属性页，并且只是在需要时才进行创建。由于这个原因，当属性表格首次被创建时，每个具体的属性页将不会得到 WM_INITDIALOG 消息，并且，如果该属性页从不会被激活，那么它的对话框过程将无法得到 WM_INITDIALOG 消息。

与常规对话框的对话过程一样，属性页对话过程接收有关它的控制的 WM_COMMAND 通知消息。但是如果没有常规对话框的 OK 和 Cancel 按钮，你就不应该从属性页对话过程中调用 EndDialog 来关闭属性页或者属性表格，而应该在用户关闭属性表格时，让属性表格来清除每一页。

12.3.2.1 WM_NOTIFY消息

属性表格发送 WM_NOTIFY 消息来通知对话过程有关属性表格状态的重要变化，这个消息标识一个属性页何时被激活以及何时失活(deactivate)。对于需要为控制提供验证的属性页，失活通知提供了检查用户输入值的机会，当出现非法值时，还有机会阻止属性页失活。当用户按下 OK 或者 Apply 按钮请求使所作的修改生效，或者当用户按下 Cancel 按钮请求忽略所作的修改时，还会发送一条 WM_NOTIFY 消息。当属性表格被创建为向导时，还有另一组 WM_NOTIFY 消息通知对话过程发生在向导按钮 Back、 Next 或者 Finish 上的单击。

WM_NOTIFY 消息的 lParam 包含一个指向 NMHDR 的指针；这个数据结构中的 code 成员标识发送通知的原因。下面的代码框架处理了所有可用的属性表格通知：

```
case WM_NOTIFY :
     {
     LPNMHDR pnmh = (LPNMHDR) lParam ;
     switch (pnmh->code)
          {
          case PSN_SETACTIVATE :
               [property page activated]
               break ;

          case PSN_KILLACTIVE :
               [property page deactivated]
               break ;

          case PSN_APPLY :
               [user clicked OK or Apply]
               break ;

          case PSN_RESET :
               [user clicked Cancel]
               break ;
```

```
        case PSN_QUERYCANCEL :
             [user clicked Cancel -- Ok?]
             break ;

        case PSN_HELP :
             [user clicked Help]
             break ;

        case PSN_WIZBACK :
             [user clicked Back on wizard]
             break ;

        case PSN_WIZNEXT :
             [user clicked Next on wizard]
             break ;

        case PSN_WIZFINISH :
             [user clicked Finish on wizard]
             break ;
        }
   }
```

各个属性页通知的含义总结在下表中：

通知码	用户动作	描述
PSN_APPLY	用户单击 OK 按钮(关闭属性页）或者 Apply 按钮（不关闭属性页）。活动以及非活动的页获得这个通知，但是只在该页创建之后。	接受所做的修改
PSN_HELP	Help 按钮	
PSN_KILLACTIVE	用户关闭属性页，或者移动到另一个属性页，或者单击了 Apply 按钮。	验证页的内容的通知；当检查到非法值时可以阻止属性页的关闭
PSN_QUERYCANCEL	Cancel 按钮	询问是否可以允许用户取消属性表格。
PSN_RESET	Cancel 按钮	关于忽略最后一次按下 Apply 按钮以来所做的所有动作的通知。
PSN_SETACITVE	用户启动属性页，并且当前页开始时是活动的，或者用户选中了一个新的活动页，或者用户按下了 Apply 按钮（该按钮使当前页失活然后重新激活）。	通知一个属性页是活动的。
PSN_WIZBACK	用户单击了向导上的 Back 按钮	
PSN_WIZFINISH	用户单击了向导上的 Finish 按钮	
PSN_WIZNEXT	用户单击了向导上的 Next 按钮	

这些通知中最重要的是 PSN_APPLY，它意味着用户单击了 OK 或者 Apply 按钮，所请求的修改应该被接受。这个通知取代了普通对话框的带有 IDOK 控制的 ID 的 WM_COMMAND 消息。不过，与普通对话框不同，你不能在属性页对话框中调用 EndDialog（因为这样做将导致属性页的消失），而应该让属性表格自己来管理其中单个属性页的创建和释放。

当按下两个按钮 OK 或者 Apply 中的任意一个时，都将发送 PSN_APPLY，并且在大多数情况下你只是验证修改后的值，而不会试图区分这两个按钮。然而，在有些情况下对这两个按钮的处理是不同的：对于非模态属性表格，你必须在按下 OK 时释放属性表格。下面显示了如何做到这一点：

```
// Check for modeless property sheet
if (IsWindowEnabled (hwndMain))
     {
     LPPSHNOTIFY psh = (LPPSHNOTIFY) lParam ;
     HWND hwndPropSheet ;

     // Click Apply button means no destroy
     if (pnmh->code == PSN_APPLY && psh->lParam == 0)
          return TRUE ;

     // Clicking OK or Cancel, destroy property sheet
     hwndPropSheet = GetParent (hwndDlg) ;
     DestroyWindow (hwndPropSheet) ;
     hwndModeless = NULL ;
     }
```

PSN_APPLY 和 PSN_RESET 通知被广播给属性表格中每个活动的和不活动的属性页的对话框过程，但是有一个限制：属性页必须是被创建了的。由于属性表格采用了一种省事儿的方法来创建单个的属性页，只有当用户首次激活一页时才创建该页，从而有可能在属性表格中存在的某个属性页对话框过程没有收到 WM_NOTIFY 消息。在实践中，这意味着一个属性页对话将总是在收到一条 WM_NOTIFY 消息之前先接收到一条 WM_INITDIALOG 消息，因此它可以初始化它的所有控制。

12.3.2.2 控制属性表格

你通过给属性表格发消息来控制它，为此你需要属性表格的句柄。属性页对话过程接收属性页的句柄，这些属性页是属性表格的子窗口。接收属性表格的句柄要求对 GetParent 的调用：

```
// 获得属性表格的句柄
HWND hwndPropSheet = GetParent (hwndDlg) ;
```

你以这个句柄为参数调用 SendMessage，使用 PRSHT.H 中提供的属性表格宏可以很容易地完成。

下表总结了可用的属性表格消息：

类别	消息	描述
激活页	PSM_SETCURSEL	激活页的通知，页由从 0 开始的索引或者属性页句柄来标识。
	PSM_SETCURSELID	激活页的通知，页由页资源 ID 标识。
按钮状态	PSM_APPLY	模拟用户按下 Apply 按钮，这将使所有现存的页收到一条带有 PSN_APPLY 通知码的 WM_NOTIFY 消息。
	PSM_CANCELTOCLOSE	禁止 Cancel 按钮并将 OK 按钮的文本改变为 Close
	PSM_CHANGED	为当前页设置“脏标志”。如果任何一页被设置了脏标志，则允许 Apply 按钮；否则，该按钮将被禁止。
	PSM_UNCHANGED	为当前页清除“脏标志”；参见上面的 PMS_CHANGED 。
	PSM_PRESSBUTTON	模拟用户按下属性表格按钮。
	PSM_SETFINISHTEXT	启用向导上的 Finish 按钮并设置它的文本。
	PSM_SETWIZBUTTONS	启用向导按钮。
编辑页列表	PSM_ADDPAGE	在属性表格中增加属性页。
	PSM_REMOVEPAGE	从属性表格中删除属性页。
查询	PSM_GETTABCONTROL	获取 Tab 控制的句柄。
	PSM_GETCURRENTPAGEHWND	获取当前属性页窗口的窗口句柄。
	PSM_QUERYSIBLINGS	当接收到这条消息时，属性表格对所有现存的属性页发送一条 PSM_QUERYSIBLINGS 消息。（“现存的”意味着用户访问过该页，从而使该页的对话过程收到一条 WM_INITDIALOG 消息。）
修改要求重启动	PSM_REBOOTSYSTEM	设置标志以便让 PropertySheet 函数返回 ID_PSREBOOTSYSTEM 。
	PSM_RESTARTWINDOWS	设置标志以便让 PrportyeSheet 函数返回 ID_PSRESTARTWINDOWS 。
杂类	PSM_ISDIALOGMESSAGE	允许非模态属性表格的键盘接口（参见从 P590 开始的讨论）。
	PSM_SETTITLE	设置属性表格上的标题。

12.3.2.3 启用和禁止Apply按钮

当属性表格被创建时， Apply 按钮是被禁用的，必须当用户修改了某些数据时，从单

个属性页的对话框过程中显式地启用。通过发送一条 PSM_CHANGED 消息来通知属性表格数据已被修改；通过对属性表格发送一条 PSM_UNCHANGES 消息可以取消修改（从而禁止 Apply 按钮）。使用 PRSHT.H 消息宏，下面是如何向一个属性表格发送这个消息：

```
hwndSheet = GetParent (hwndDlg) ;
PropSheet_Changed (hwndSheet, hwndDlg) ;
```

属性表格为每个属性页保持一个“已修改”位，这样，当任何一个属性页确定它的内容已被修改时 Apply 按钮都会被启用，而不论在某一既定时刻显示的究竟是哪一页。

12.3.2.4 使修改生效

当用户按下 OK 或者 Apply 按钮时，意味着应该处理所作的修改了。这两个按钮事件应该以相同的方式来处理，虽然按下 OK 按钮将关闭属性表格，而按下 Apply 按钮却不会。将这两种修改的处理组合为一个操作的方法是给父窗口发送一条消息来请求使所作的修改生效，这与在典型的 Windows 程序中处理模态对话框的方法不同，后一种方法是在开始主程序的任何修改之前等待 DialogBox 函数返回。事实上，这里的方法更像一个非模态对话框中的修改处理方式。

来自 OK 按钮和 Apply 按钮的通知是一样的。现有属性页——用户访问过的并且其对话过程收到过 WM_INITDIALOG 消息的页——的对话过程将得到一条带有 PSN_APPLY 通知码的 WM_NOTIFY 消息，合理的响应是记下发生的所有修改并给父窗口发送一条消息。

为了使属性表格的行为更像模态对话框，可以创建带有 PSH_NOAPPLYNOW 风格标志的属性表格，隐藏 Apply 按钮。（“ Apply Now ”是以前的按钮标题。）这样使修改生效需要监视由 PropertySheet 返回的值：按下 OK 返回 TRUE 值，按下 Cancel 返回 FALSE 值。

12.3.2.5 属性表格帮助

有两种技术用于请求一个属性表格上的帮助：Help 按钮和 F1 键。Help 按钮只出现在使用 PSH_HASHELP 标志创建的属性表格上，而且在带有 PSP_HASHELP 标志的属性页被激活之前，Help 按钮将保持被禁止状态。按动 Help 按钮导致一条带有 PSN_HELP 通知码的 WM_NOTIFY 消息被发送给属性页对话过程。你可以假定对这条消息的响应应该是显示一个帮助窗口，其中包含关于使用某个属性页的提示。

每个 Win32 窗口都支持 F1 帮助键。缺省的窗口过程对不处理的 F1 键的响应是给窗口的父窗口发送一条 WM_HELP 消息，因此，要启用 F1 键帮助支持，除了处理 WM_HELP 消息外，不需要什么特别的处理。

12.3.2.6 操纵非模态属性表格

正如大多数对话框都是模态对话框一样，大多数的属性表格也都是模态的。如果你想要非模态的属性表格，需要进行一些额外的修改。首先，为了获得一个非模态对话框，应在 PROPSHEETHEADER 结构中指定 PSH_MODELESS 标志；此外的修改和你创建非模态对话框时所需要的相同。

为了启用非模态对话框的键盘界面，你可以调用 IsDialogMessage，这个函数过滤键盘消息的消息循环。当收到一条键盘消息时，拥有键盘焦点的控制接收一条

WM_GETDLGCODE 消息，该消息实际上是询问这个控制，“你想要哪些类型的键盘输入？”根据回答，控制可能会获得键盘输入或者（在响应 Tab 键、Alt+助记符或者箭头键时）将焦点移到对话框的 Tab 顺序中的下一个非静态控制中。

对于非模态的属性表格，应该给属性表格的父窗口发送一条 PSM_ISDIALOGMESSAGE 消息，而不是调用 IsDialogMessage，因此，具有非模态属性表格的程序将有一个如下所示的消息循环：

```
while (GetMessage (&msg, NULL, 0, 0))
     {
     if ((hwndModeless) &&
         (!(PropSheet_IsDialogMessage (hwndModeless, &msg))))
          continue ;

     TranslateMessage (&msg) ;
     DispatchMessage (&msg) ;
     }
```

这段代码假设，当存在非模态表格时，hwndModeless 为其句柄，否则，hwndModeless 为 NULL。

12.3.2.7 设置向导按钮

当创建向导而不是属性表格时，每个属性页的对话过程必须控制向导按钮。属性表格有 OK 和 Cancel 按钮，而向导则有 Back 和 Next 按钮。两个按钮都保持启用状态，除非你明确地禁止了它们。并且在向导的最后一页中，用户会看到 Finish 按钮代替 Next 按钮。

要控制向导按钮的设置，发送一条 PSM_SETWIZBUTTONS 消息，并将消息的 lParam 设置为三个值的组合：PSWIZB_NEXT、PSWIZB_BACK 和 PSWIZB_FINISH。发送这条消息的准确时间是当向导页活动时，这时，属性页对话框知道这一事件，因为对话过程收到了以 WM_NOTIFY 消息的形式给出的 PSN_SETACTIVE 通知。为了响应 PSN_SETACTIVE 通知，向导可以通过返回-1 来阻止一页被激活。另一种选择是，通过返回特定向导页的资源 ID，该向导页成为下一个将被激活的页，否则，返回 0 将允许另一页被激活。

当向导的第一页被激活时，通过发送 lParam 为 PSWIZE_NEXT 的 PSM_SETWIZBUTTONS 消息来禁止 Back 按钮。PropSheet_SetWizButtons 宏提供了发送这条消息的一种方法：

```
// 启用向导页上的 Next 按钮
if (bWizard && pnmh->code == PSN_SETACTIVE)
    {
    HWND hwndSheet = GetParent (hwndDlg) ;
    PropSheet_SetWizButton (hwndSheet, PSWIZB_NEXT) ;
    }
```

在向导的最后一页，启用 Back 按钮，并且，通过发送一条 PSM_SETWIZBUTTONS 消息，将消息中的 lParam 设置为 PSWIZB_BACK | PSWIZB_FINISH，将 Next 按钮的文本改变为"Finish"。

```
// 允许向导页上的 Back 和 Finish 按钮
if (bWizard && pnmh->code == PSN_SETACTIVE)
    {
    HWND hwndSheet = GetParent (hwndDlg) ;
    ProperSheet_SetWizButton (hwndSheet, PSWIZB_BACK | PSWIZB_FINISH) ;
    }
```

12.3.3 PROPERTY 程序

图 12-11 和图 12-12 显示了属性表格示例程序 PROPERTY 创建的属性表格上的两个属性页。这两个属性页合起来，定义了所有可以传递给 CreateWindowEx 函数的通用风格标志（WS_和 WS_EX_）。PROPERTY 的主窗口显示了两个窗口：一个（“First Window”）是使用属性表格中编辑的风格标志来创建的；另一个（“Second Window”）用于显示某些窗口剪裁风格标志（WS_CLIPSIBLINGS）的效果，这些标志在两个子窗口共享相同的象素时，可以防止“兄弟相争”。除了演示创建属性表格的基本技术外，PROPERTY 还演示了大部分窗口创建风格标志的功能。PROPERTY 的源程序清单在图 12-13 中给出。

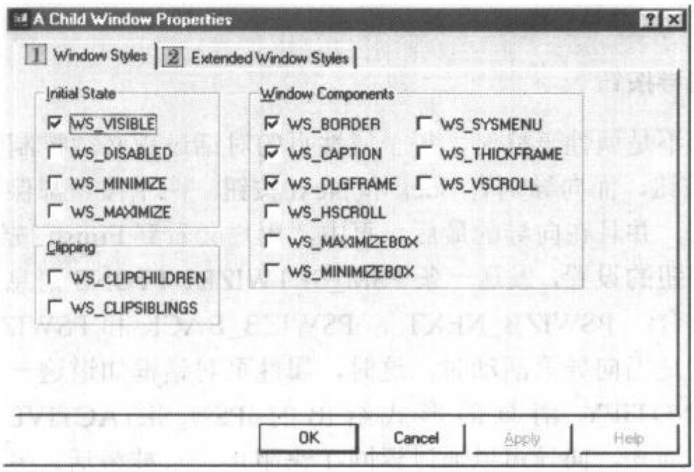


图 12-11　窗口风格属性页

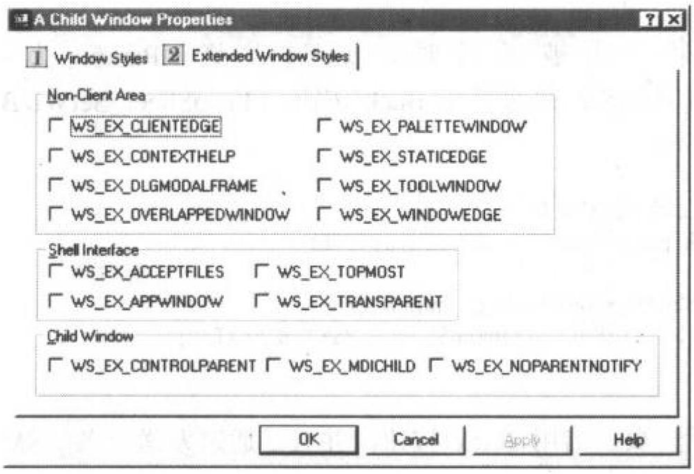


图 12-12　扩展的窗口风格属性页

PROPERTY.MAK

```
#-----------------------
# PROPERTY.MAK make file
#-----------------------

property.exe : property.obj helper.obj notify.obj \
               page1.obj page2.obj sheet.obj property.res
     $(LINKER) $(GUIFLAGS) -OUT:property.exe property.obj \
     helper.obj notify.obj page1.obj page2.obj sheet.obj \
     property.res $(GUILIBS)

property.obj : property.c comcthlp.h property.h
     $(CC) $(CFLAGS) property.c

helper.obj : helper.c property.h
     $(CC) $(CFLAGS) helper.c

notify.obj : notify.c notify.h
     $(CC) $(CFLAGS) notify.c

page1.obj : page1.c helper.h notify.h property.h
     $(CC) $(CFLAGS) page1.c

page2.obj : page2.c helper.h notify.h property.h
     $(CC) $(CFLAGS) page2.c

sheet.obj : sheet.c comcthlp.h property.h
     $(CC) $(CFLAGS) sheet.c

property.res : property.rc property.ico
     $(RC) $(RCVARS) property.rc
```

PROPERTY.H

```
// Menu item identifiers
#define IDM_OVERLAPPED    100
#define IDM_POPUP         101
#define IDM_CHILD         102
#define IDM_WIZARD        200
#define IDM_HASHELP       201
#define IDM_MODELESS      202
#define IDM_MULTILINETABS 203
#define IDM_NOAPPLYNOW    204
#define IDM_PROPTITLE     205
#define IDM_RTLREADING    206

// Dialog template IDs
#define IDD_STYLES                  101
#define IDD_EXSTYLES                102

// Icon IDs
#define IDI_PAGE1                   103
#define IDI_PAGE2                   104

// Dialog Control IDs
```

```
#define IDC_BORDER                  1000
#define IDC_CAPTION                 1001
#define IDC_VISIBLE                 1005
#define IDC_DISABLED                1006
#define IDC_DLGFRAME                1007
#define IDC_MINIMIZE                1008
#define IDC_MAXIMIZE                1009
#define IDC_HSCROLL                 1010
#define IDC_MAXIMIZEBOX             1011
#define IDC_MINIMIZEBOX             1012
#define IDC_SYSMENU                 1013
#define IDC_THICKFRAME              1014
#define IDC_VSCROLL                 1015
#define IDC_CLIPCHILDREN            1016
#define IDC_CLIPSIBLINGS            1017
#define IDC_CLIENTEDGE              1018
#define IDC_CONTEXTHELP             1019
#define IDC_DLGMODALFRAME           1020
#define IDC_EXOVERLAPPED            1021
#define IDC_PALETTE                 1022
#define IDC_STATICEDGE              1023
#define IDC_TOOLWINDOW              1024
#define IDC_WINDOWEDGE              1025
#define IDC_ACCEPTFILES             1026
#define IDC_APPWINDOW               1027
#define IDC_TOPMOST                 1028
#define IDC_TRANSPARENT             1029
#define IDC_CONTROLPARENT           1030
#define IDC_MDICHILD                1031
#define IDC_NOPARENTNOTIFY          1032
#define IDM_WINDOW_PROPERTIES       40001
#define IDC_STATIC                  -1
#define IDI_APP                     1000

// Private message
#define PM_CREATEWINDOW   WM_APP

// Property Sheet Functions (in SHEET.C)
BOOL CreatePropertySheet (HWND hwndParent) ;

// Property Page Functions (in PAGE1.C and PAGE2.C)
UINT CALLBACK StylePageProc (HWND, UINT, LPPROPSHEETPAGE) ;
BOOL CALLBACK StyleDlgProc (HWND, UINT, WPARAM, LPARAM) ;
UINT CALLBACK ExStylePageProc (HWND, UINT, LPPROPSHEETPAGE) ;
BOOL CALLBACK ExStyleDlgProc (HWND, UINT, WPARAM, LPARAM) ;
```

HELPER.H

```
void SetButtonCheck (HWND hwndDlg, int CtrlID, BOOL bCheck) ;
BOOL QueryButtonCheck (HWND hwndDlg, int CtrlID) ;
```

PROPERTY.C

```
/*------------------------------------
   PROPERTY.C -- Property sheet example
                 (c) Paul Yao, 1996
  ------------------------------------*/
```

```
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "property.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[]  = "Property Sheet" ;
HINSTANCE hInst ;
HWND hwndMain = NULL ;
HWND hwndChild = NULL ;
HWND hwndModeless = NULL ;

HICON hiconApp ;   // Application icon
BOOL bWizard ;     // Flag whether PSH_WIZARD is set

// Values modified by property sheet
DWORD dwChildStyle = WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION ;
DWORD dwChildExStyle = 0L ;

// Value modified by menu item selection
DWORD dwSheetStyles = PSH_PROPTITLE ;

//-------------------------------------------------------------------
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX wc ;

     hInst = hInstance ;
     hiconApp = LoadIcon (hInst, MAKEINTRESOURCE (IDI_APP)) ;

     ZeroMemory (&wc, sizeof (WNDCLASSEX)) ;
     wc.cbSize         = sizeof (wc) ;
     wc.lpszClassName = "MAIN" ;
     wc.hInstance      = hInstance ;
     wc.lpfnWndProc    = WndProc ;
     wc.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wc.hIcon          = hiconApp ;
     wc.lpszMenuName   = "MAIN" ;
     wc.hbrBackground = (HBRUSH) (COLOR_APPWORKSPACE + 1) ;
     wc.hIconSm        = hiconApp ;

     RegisterClassEx (&wc) ;

     wc.lpszClassName = "CHILD" ;
     wc.lpfnWndProc    = DefWindowProc ;
     wc.hCursor        = LoadCursor (NULL, IDC_IBEAM) ;
     wc.hIcon          = NULL ;
     wc.lpszMenuName   = NULL ;
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wc.hIconSm        = NULL ;
     RegisterClassEx (&wc) ;
```

```
     hwndMain =
     hwnd = CreateWindowEx (WS_EX_OVERLAPPEDWINDOW, "MAIN",
                            szAppName, WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, cmdShow) ;
     UpdateWindow (hwnd) ;

     InitCommonControls () ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          if ((hwndModeless) &&
              (!(PropSheet_IsDialogMessage (hwndModeless, &msg))))
              continue ;

          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

//-------------------------------------------------------------------
void MenuCheckMark (HMENU hmenu, int id, BOOL bCheck)
     {
     int iState ;
     iState = (bCheck) ? MF_CHECKED : MF_UNCHECKED ;
     CheckMenuItem (hmenu, id, iState) ;
     }

//-------------------------------------------------------------------
void static FlipFlag (LPDWORD dwStyle, DWORD flag)
     {
     if (*dwStyle & flag)   // Flag on -- turn off
          {
          *dwStyle &= (~flag) ;
          }
     else                   // Flag off -- turn on
          {
          *dwStyle *= flag ;
          }
     }

//-------------------------------------------------------------------
LRESULT CALLBACK
WndProc (HWND hwnd, UINT mMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (mMsg)
          {
          case WM_CREATE :
               hwndChild = CreateWindowEx (dwChildExStyle, "CHILD",
                                "First Window", dwChildStyle,
                                0, 0, 0, 0, hwnd, (HMENU) 1,
                                          hInst, NULL) ;
```

```
          CreateWindowEx (dwChildExStyle, "CHILD", "Second Window",
                          WS_CLIPSIBLINGS * dwChildStyle,
                          10, 10, 200, 50, hwnd,
                          (HMENU) 2, hInst, NULL) ;
          return 0 ;

     case WM_COMMAND :
          {
          switch (LOWORD (wParam))
               {
               case IDM_WINDOW_PROPERTIES :
                    {
                    BOOL bRet ;

                    // If modeless, activate existing property sheet
                    if (hwndModeless)
                         {
                         SetActiveWindow (hwndModeless) ;
                         break ;
                         }

                    // Are we creating a wizard?
                    bWizard = (dwSheetStyles & PSH_WIZARD) ;

                    // Create actual property sheet
                    bRet = CreatePropertySheet (hwnd) ;

                    // Store handle if modeless
                    if (dwSheetStyles & PSH_MODELESS)
                         {
                         hwndModeless = (HWND) bRet ;
                         break ;
                         }
                    break ;
                    }

               case IDM_WIZARD :
                    FlipFlag (&dwSheetStyles, PSH_WIZARD) ;
                    break ;

               case IDM_HASHELP :
                    FlipFlag (&dwSheetStyles, PSH_HASHELP) ;
                    break ;

               case IDM_MODELESS :
                    FlipFlag (&dwSheetStyles, PSH_MODELESS) ;
                    break ;

               case IDM_NOAPPLYNOW :
                    FlipFlag (&dwSheetStyles, PSH_NOAPPLYNOW) ;
                    break ;

               case IDM_PROPTITLE :
                    FlipFlag (&dwSheetStyles, PSH_PROPTITLE) ;
                    break ;
```

```
               case IDM_RTLREADING :
                    FlipFlag (&dwSheetStyles, PSH_RTLREADING) ;
                    break ;
               }
               return 0 ;
          }

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;

     case WM_INITMENU :
          {
          BOOL bCheck ;
          HMENU hmenu = (HMENU) wParam ;

          bCheck = (dwSheetStyles & PSH_WIZARD) ;
          MenuCheckMark (hmenu, IDM_WIZARD, bCheck) ;

          bCheck = (dwSheetStyles & PSH_HASHELP) ;
          MenuCheckMark (hmenu, IDM_HASHELP, bCheck) ;

          bCheck = (dwSheetStyles & PSH_MODELESS) ;
          MenuCheckMark (hmenu, IDM_MODELESS, bCheck) ;

          bCheck = (dwSheetStyles & PSH_NOAPPLYNOW) ;
          MenuCheckMark (hmenu, IDM_NOAPPLYNOW, bCheck) ;

          bCheck = (dwSheetStyles & PSH_PROPTITLE) ;
          MenuCheckMark (hmenu, IDM_PROPTITLE, bCheck) ;

          bCheck = (dwSheetStyles & PSH_RTLREADING) ;
          MenuCheckMark (hmenu, IDM_RTLREADING, bCheck) ;

          return 0 ;
          }

     case WM_SETFOCUS :
          SetFocus (hwndChild) ;
          return 0 ;

     case WM_SIZE :
          {
          int cxWidth  = LOWORD (lParam) ;
          int cyHeight = HIWORD (lParam) ;
          int x, y, cx, cy ;

          x  = cxWidth  / 4 ;
          cx = cxWidth  / 2 ;
          y  = cyHeight / 4 ;
          cy = cyHeight / 2 ;

          MoveWindow (hwndChild, x, y, cx, cy, TRUE) ;
          return 0 ;
          }

     case PM_CREATEWINDOW :
```

```
            {
            RECT rClient ;
            LPARAM l ;

            DestroyWindow (hwndChild) ;
            hwndChild = CreateWindowEx (dwChildExStyle, "CHILD",
                                        "First Window", dwChildStyle,
                                        0, 0, 0, 0, hwnd, (HMENU) 1,
                                               hInst, NULL) ;

            // Send ourselves a WM_SIZE to resize child window
            GetClientRect (hwnd, &rClient) ;
            l = MAKELPARAM (rClient.right, rClient.bottom) ;
            SendMessage (hwnd, WM_SIZE, 0, l) ;
            return 0 ;
            }

        default :
            return (DefWindowProc (hwnd, mMsg, wParam, lParam)) ;
        }
    }
```

SHEET.C

```
/*-------------------------------------------
   SHEET.C -- Property sheet page functions
              (c) Paul Yao, 1996
  -------------------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "comcthlp.h"
#include "property.h"

extern DWORD dwChildStyle ;
extern DWORD dwChildExStyle ;
extern DWORD dwSheetStyles ;
extern char szAppName[] ;
extern HINSTANCE hInst ;
extern HICON hiconApp ;

//-------------------------------------------------------------------
int CALLBACK PropSheetProc (HWND hwndDlg, UINT uMsg, LPARAM lParam)
    {
    switch (uMsg)
        {
        case PSCB_INITIALIZED :
            // Process PSCB_INITIALIZED
            break ;

        case PSCB_PRECREATE :
            // Process PSCB_PRECREATE
            break ;
        default :
            // Unknown message
            break ;
        }
```

```
     return 0 ;
     }

//----------------------------------------------------------------------
BOOL CreatePropertySheet (HWND hwndParent)
     {
     PROPSHEETHEADER pshead ;
     PROPSHEETPAGE   pspage[2] ;

     // Initialize property sheet HEADER data
     ZeroMemory (&pshead, sizeof (PROPSHEETHEADER)) ;
     pshead.dwSize  = sizeof (PROPSHEETHEADER) ;
     pshead.dwFlags = dwSheetStyles      |
                      PSH_PROPSHEETPAGE |
                      PSH_USECALLBACK   |
                      PSH_USEHICON ;
     pshead.hwndParent  = hwndParent ;
     pshead.hInstance   = hInst ;
     pshead.hIcon       = hiconApp ;
     pshead.pszCaption  = "A Child Window" ;
     pshead.nPages      = 2 ;
     pshead.nStartPage  = 0 ;
     pshead.ppsp        = pspage ;
     pshead.pfnCallback = PropSheetProc ;

     // Zero out property PAGE data
     ZeroMemory (&pspage, 2 * sizeof (PROPSHEETPAGE)) ;

     // PAGE 1 -- window style page
     pspage[0].dwSize      = sizeof (PROPSHEETPAGE) ;
     pspage[0].dwFlags     = PSP_USECALLBACK | PSP_USEICONID ;
     pspage[0].hInstance   = hInst ;
     pspage[0].pszTemplate = MAKEINTRESOURCE (IDD_STYLES) ;
     pspage[0].pszIcon     = MAKEINTRESOURCE (IDI_PAGE1) ;
     pspage[0].pfnDlgProc  = StyleDlgProc ;
     pspage[0].lParam      = (LPARAM) &dwChildStyle ;
     pspage[0].pfnCallback = StylePageProc ;

     // PAGE 2 -- extended window style page
     pspage[1].dwSize      = sizeof (PROPSHEETPAGE) ;
     pspage[1].dwFlags     = PSP_USECALLBACK | PSP_USEICONID |
                             PSP_HASHELP ;
     pspage[1].hInstance   = hInst ;
     pspage[1].pszTemplate = MAKEINTRESOURCE (IDD_EXSTYLES) ;
     pspage[1].pszIcon     = MAKEINTRESOURCE (IDI_PAGE2) ;
     pspage[1].pfnDlgProc  = ExStyleDlgProc ;
     pspage[1].lParam      = (LPARAM) &dwChildExStyle ;
     pspage[1].pfnCallback = ExStylePageProc ;

     // --------- Create & display property sheet ---------

     return PropertySheet (&pshead) ;
     }
```

PAGE1.C

```
/*---------------------------------
   PAGE1.C -- Property sheet page 1
              (c) Paul Yao, 1996
  ---------------------------------*/
#include <windows.h>
#include <prsht.h>
#include "property.h"
#include "notify.h"
#include "helper.h"

static LPDWORD pTheStyles ;
extern BOOL bWizard ;
extern HWND hwndMain ;
extern HWND hwndModeless ;

DWORD FetchStyles (HWND hwndDlg) ;

//-------------------------------------------------------------------
UINT CALLBACK
StylePageProc (HWND  hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
     {
     switch (uMsg)
          {
          case PSPCB_CREATE :
               // Store pointer to style data
               pTheStyles = (LPDWORD) ppsp->lParam ;
               return TRUE ;

          case PSPCB_RELEASE :
               return 0 ;
          }

     return 0 ;
     }

//-------------------------------------------------------------------
BOOL CALLBACK
StyleDlgProc (HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               {
               BOOL bCheck ;
               DWORD dwOrigStyle = *pTheStyles ;

               bCheck = (dwOrigStyle & WS_VISIBLE) ;
               SetButtonCheck (hwndDlg, IDC_VISIBLE, bCheck) ;

               bCheck = (dwOrigStyle & WS_DISABLED) ;
               SetButtonCheck (hwndDlg, IDC_DISABLED, bCheck) ;

               bCheck = (dwOrigStyle & WS_MINIMIZE) ;
               SetButtonCheck (hwndDlg, IDC_MINIMIZE, bCheck) ;

               bCheck = (dwOrigStyle & WS_MAXIMIZE) ;
               SetButtonCheck (hwndDlg, IDC_MAXIMIZE, bCheck) ;
```

```
          bCheck = (dwOrigStyle & WS_CLIPCHILDREN) ;
          SetButtonCheck (hwndDlg, IDC_CLIPCHILDREN, bCheck) ;

          bCheck = (dwOrigStyle & WS_CLIPSIBLINGS) ;
          SetButtonCheck (hwndDlg, IDC_CLIPSIBLINGS, bCheck) ;

          bCheck = (dwOrigStyle & WS_BORDER) ;
          SetButtonCheck (hwndDlg, IDC_BORDER, bCheck) ;

          bCheck = (dwOrigStyle & WS_CAPTION) ;
          SetButtonCheck (hwndDlg, IDC_CAPTION, bCheck) ;

          bCheck = (dwOrigStyle & WS_DLGFRAME) ;
          SetButtonCheck (hwndDlg, IDC_DLGFRAME, bCheck) ;

          bCheck = (dwOrigStyle & WS_HSCROLL) ;
          SetButtonCheck (hwndDlg, IDC_HSCROLL, bCheck) ;

          bCheck = (dwOrigStyle & WS_MAXIMIZEBOX) ;
          SetButtonCheck (hwndDlg, IDC_MAXIMIZEBOX, bCheck) ;

          bCheck = (dwOrigStyle & WS_MINIMIZEBOX) ;
          SetButtonCheck (hwndDlg, IDC_MINIMIZEBOX, bCheck) ;

          bCheck = (dwOrigStyle & WS_SYSMENU) ;
          SetButtonCheck (hwndDlg, IDC_SYSMENU, bCheck) ;

          bCheck = (dwOrigStyle & WS_THICKFRAME) ;
          SetButtonCheck (hwndDlg, IDC_THICKFRAME, bCheck) ;

          bCheck = (dwOrigStyle & WS_VSCROLL) ;
          SetButtonCheck (hwndDlg, IDC_VSCROLL, bCheck) ;

          return TRUE ;
     }

     case WM_COMMAND :
          {
          WORD wNotifyCode = HIWORD (wParam) ;
          WORD wID = LOWORD (wParam) ;
          HWND hwndSheet ;

          switch (wID)
               {
               case IDC_VISIBLE :
               case IDC_DISABLED :
               case IDC_MINIMIZE :
               case IDC_MAXIMIZE :
               case IDC_CLIPCHILDREN :
               case IDC_CLIPSIBLINGS :
               case IDC_BORDER :
               case IDC_CAPTION :
               case IDC_DLGFRAME :
               case IDC_HSCROLL :
               case IDC_MAXIMIZEBOX :
               case IDC_MINIMIZEBOX :
```

```
            case IDC_SYSMENU :
            case IDC_THICKFRAME :
            case IDC_VSCROLL :
                 hwndSheet = GetParent (hwndDlg) ;
                 PropSheet_Changed (hwndSheet, hwndDlg) ;
                 break ;
            }
       return TRUE ;
       }

  case WM_HELP :
       // Catch F1 key strike
       MessageBox (hwndDlg, "WM_HELP Message Received",
                   "StyleDlgProc", MB_OK) ;
       return TRUE ;

  case WM_NOTIFY :
       {
       LPNMHDR pnmh = (LPNMHDR) lParam ;

       // Handle OK and Apply buttons
       if (pnmh->code == PSN_APPLY)
            {
            HWND hwndPS ;
            HWND hwndActive ;

            // Overwrite current style value
            *pTheStyles = FetchStyles (hwndDlg) ;

            // Tell main window to re-create child window
            hwndPS = GetParent (hwndDlg) ;
            hwndActive = PropSheet_GetCurrentPageHwnd (hwndPS) ;

            // Only re-create if we're the active page
            if (hwndDlg == hwndActive)
                 PostMessage (hwndMain, PM_CREATEWINDOW, 0, 0L) ;
            }

       // Destroy modeless dialog on OK or Cancel
       if ((IsWindowEnabled (hwndMain)) &&
           (pnmh->code == PSN_APPLY || pnmh->code == PSN_RESET))
            {
            LPPSHNOTIFY psh = (LPPSHNOTIFY) lParam ;
            HWND hwndPropSheet ;

            // Ignore Apply button
            if (pnmh->code == PSN_APPLY && psh->lParam == 0)
                 return TRUE ;

            // Clicking OK or Cancel, destroy property sheet
            hwndPropSheet = GetParent (hwndDlg) ;
            DestroyWindow (hwndPropSheet) ;
            hwndModeless = NULL ;
            }

       // Enable Next button on wizard page
       if (bWizard && pnmh->code == PSN_SETACTIVE)
```

```
                    {
                    HWND hwndSheet = GetParent (hwndDlg) ;
                    PropSheet_SetWizButtons (hwndSheet, PSWIZB_NEXT) ;
                    }
               return TRUE ;
               }

          default :
               return FALSE ;
          }
     }

//----------------------------------------------------------------
DWORD FetchStyles (HWND hwndDlg)
     {
     DWORD dwStyle = WS_CHILD ;

     if (QueryButtonCheck (hwndDlg, IDC_VISIBLE))
          {
          dwStyle *= WS_VISIBLE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_DISABLED))
          {
          dwStyle *= WS_DISABLED ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_MINIMIZE))
          {
          dwStyle *= WS_MINIMIZE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_MAXIMIZE))
          {
          dwStyle *= WS_MAXIMIZE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_CLIPCHILDREN))
          {
          dwStyle *= WS_CLIPCHILDREN ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_CLIPSIBLINGS))
          {
          dwStyle *= WS_CLIPSIBLINGS ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_BORDER))
          {
          dwStyle *= WS_BORDER ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_CAPTION))
          {
          dwStyle *= WS_CAPTION ;
          }
```

```
    if (QueryButtonCheck (hwndDlg, IDC_DLGFRAME))
        {
        dwStyle *= WS_DLGFRAME ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_HSCROLL))
        {
        dwStyle *= WS_HSCROLL ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_MAXIMIZEBOX))
        {
        dwStyle *= WS_MAXIMIZEBOX ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_MINIMIZEBOX))
        {
        dwStyle *= WS_MINIMIZEBOX ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_SYSMENU))
        {
        dwStyle *= WS_SYSMENU ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_THICKFRAME))
        {
        dwStyle *= WS_THICKFRAME ;
        }

    if (QueryButtonCheck (hwndDlg, IDC_VSCROLL))
        {
        dwStyle *= WS_VSCROLL ;
        }

    return dwStyle ;
    }
```

PAGE2.C

```
/*---------------------------------
   PAGE2.C -- Property sheet page 2
              (c) Paul Yao, 1996
  ---------------------------------*/
#include <windows.h>
#include <commctrl.h>
#include "property.h"
#include "notify.h"
#include "helper.h"

static LPDWORD pTheExStyles ;
extern BOOL bWizard ;
extern HWND hwndMain ;
extern HWND hwndModeless ;

DWORD FetchExStyles (HWND hwndDlg) ;
```

```
//------------------------------------------------------------------
UINT CALLBACK
ExStylePageProc (HWND  hwnd, UINT uMsg, LPPROPSHEETPAGE ppsp)
     {
     switch (uMsg)
          {
          case PSPCB_CREATE :
               // Store pointer to extended style data
               pTheExStyles = (LPDWORD) ppsp->lParam ;
               return TRUE ;

          case PSPCB_RELEASE :
               break ;
          }

     return 0 ;
     }

//------------------------------------------------------------------
BOOL CALLBACK
ExStyleDlgProc (HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               {
               BOOL bCheck ;
               DWORD dwOrigStyle = *pTheExStyles ;

               bCheck = (dwOrigStyle & WS_EX CLIENTEDGE) ;
               SetButtonCheck (hwndDlg, IDC_CLIENTEDGE, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_CONTEXTHELP) ;
               SetButtonCheck (hwndDlg, IDC_CONTEXTHELP, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_DLGMODALFRAME) ;
               SetButtonCheck (hwndDlg, IDC_DLGMODALFRAME, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_OVERLAPPEDWINDOW) ;
               SetButtonCheck (hwndDlg, IDC_EXOVERLAPPED, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_PALETTEWINDOW) ;
               SetButtonCheck (hwndDlg, IDC_PALETTE, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_STATICEDGE) ;
               SetButtonCheck (hwndDlg, IDC_STATICEDGE, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_TOOLWINDOW) ;
               SetButtonCheck (hwndDlg, IDC_TOOLWINDOW, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_WINDOWEDGE) ;
               SetButtonCheck (hwndDlg, IDC_WINDOWEDGE, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_ACCEPTFILES) ;
               SetButtonCheck (hwndDlg, IDC_ACCEPTFILES, bCheck) ;

               bCheck = (dwOrigStyle & WS_EX_APPWINDOW) ;
```

```
          SetButtonCheck (hwndDlg, IDC_APPWINDOW, bCheck) ;

          bCheck = (dwOrigStyle & WS_EX_TOPMOST) ;
          SetButtonCheck (hwndDlg, IDC_TOPMOST, bCheck) ;

          bCheck = (dwOrigStyle & WS_EX_TRANSPARENT) ;
          SetButtonCheck (hwndDlg, IDC_TRANSPARENT, bCheck) ;

          bCheck = (dwOrigStyle & WS_EX_CONTROLPARENT) ;
          SetButtonCheck (hwndDlg, IDC_CONTROLPARENT, bCheck) ;

          bCheck = (dwOrigStyle & WS_EX_MDICHILD) ;
          SetButtonCheck (hwndDlg, IDC_MDICHILD, bCheck) ;

          bCheck = (dwOrigStyle & WS_EX_NOPARENTNOTIFY) ;
          SetButtonCheck (hwndDlg, IDC_NOPARENTNOTIFY, bCheck) ;

          return TRUE ;
          }

     case WM_COMMAND :
          {
          WORD wNotifyCode = HIWORD (wParam) ;
          WORD wID = LOWORD (wParam) ;
          HWND hwndSheet ;

          switch (wID)
               {
               case IDC_CLIENTEDGE :
               case IDC_CONTEXTHELP :
               case IDC_DLGMODALFRAME :
               case IDC_EXOVERLAPPED :
               case IDC_PALETTE :
               case IDC_STATICEDGE :
               case IDC_TOOLWINDOW :
               case IDC_WINDOWEDGE :
               case IDC_ACCEPTFILES :
               case IDC_APPWINDOW :
               case IDC_TOPMOST :
               case IDC_TRANSPARENT :
               case IDC_CONTROLPARENT :
               case IDC_MDICHILD :
               case IDC_NOPARENTNOTIFY :
                    hwndSheet = GetParent (hwndDlg) ;
                    PropSheet_Changed (hwndSheet, hwndDlg) ;
                    break ;
               }
          return TRUE ;
          }

     case WM_HELP :
          // Catch F1 key strike
          MessageBox (hwndDlg, "WM_HELP Message Received",
                      "ExStyleDlgProc", MB_OK) ;
          return TRUE ;

     case WM_NOTIFY :
```

```
{
LPNMHDR pnmh = (LPNMHDR) lParam ;

// Handle Finish button on wizard page
if (pnmh->code == PSN_WIZFINISH)
     {
     HWND hwndPS ;

     hwndPS = GetParent (hwndDlg) ;
     PropSheet_Apply (hwndPS) ;
     return TRUE ;
     }

// Handle OK and Apply buttons
if (pnmh->code == PSN_APPLY    pnmh->code == PSN_RESET)
     {
     HWND hwndPS ;
     HWND hwndActive ;

     // Overwrite current style value
     *pTheExStyles = FetchExStyles (hwndDlg) ;

     // Tell main window to re-create child window
     hwndPS = GetParent (hwndDlg) ;
     hwndActive = PropSheet_GetCurrentPageHwnd (hwndPS) ;

     // Only re-create if we're the active page
     if (hwndDlg == hwndActive)
          PostMessage (hwndMain, PM_CREATEWINDOW, 0, 0L) ;
     }

// Destroy modeless dialog on OK or Cancel
if ((IsWindowEnabled (hwndMain)) &&
    (pnmh->code == PSN_APPLY || pnmh->code == PSN_RESET))
     {
     LPPSHNOTIFY psh = (LPPSHNOTIFY) lParam ;
     HWND hwndPropSheet ;

     // Ignore Apply button
     if (pnmh->code == PSN_APPLY && psh->lParam == 0)
          return TRUE ;

     // Clicking OK or Cancel, destroy property sheet
     hwndPropSheet = GetParent (hwndDlg) ;
     DestroyWindow (hwndPropSheet) ;
     hwndModeless = NULL ;
}

// Enable Back and Finish buttons on wizard page
if (bWizard && pnmh->code == PSN_SETACTIVE)

     {
     HWND hwndSheet = GetParent (hwndDlg) ;
     PropSheet_SetWizButtons (hwndSheet, PSWIZB_BACK |
                                         PSWIZB_FINISH) ;
     }
```

```
               // Support for Help button
               if (pnmh->code == PSN_HELP)
                    {
                    MessageBox (hwndDlg, "PSN_HELP Notification Received",
                                "ExStyleDlgProc", MB_OK) ;
                    }

               return TRUE ;
               }

          default :
               return FALSE ;
          }
     }

//-------------------------------------------------------------------
DWORD FetchExStyles (HWND hwndDlg)
     {
     DWORD dwStyle = 0L ;

     if (QueryButtonCheck (hwndDlg, IDC_CLIENTEDGE))
          {
          dwStyle *= WS_EX_CLIENTEDGE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_CONTEXTHELP))
          {
          dwStyle *= WS_EX_CONTEXTHELP ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_DLGMODALFRAME))
          {
          dwStyle *= WS_EX_DLGMODALFRAME ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_EXOVERLAPPED))
          {
          dwStyle *= WS_EX_OVERLAPPEDWINDOW ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_PALETTE))
          {
          dwStyle *= WS_EX_PALETTEWINDOW ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_STATICEDGE))
          {
          dwStyle *= WS_EX_STATICEDGE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_TOOLWINDOW))
          {
          dwStyle *= WS_EX_TOOLWINDOW ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_WINDOWEDGE))
          {
```

```
          dwStyle *= WS_EX_WINDOWEDGE ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_ACCEPTFILES))
          {
          dwStyle *= WS_EX_ACCEPTFILES ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_APPWINDOW))
          {
          dwStyle *= WS_EX_APPWINDOW ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_TOPMOST))
          {
          dwStyle *= WS_EX_TOPMOST ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_TRANSPARENT))
          {
          dwStyle *= WS_EX_TRANSPARENT ;
          }
     if (QueryButtonCheck (hwndDlg, IDC_CONTROLPARENT))
          {
          dwStyle *= WS_EX_CONTROLPARENT ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_MDICHILD))
          {
          dwStyle *= WS_EX_MDICHILD ;
          }

     if (QueryButtonCheck (hwndDlg, IDC_NOPARENTNOTIFY))
          {
          dwStyle *= WS_EX_NOPARENTNOTIFY ;
          }

     return dwStyle ;
     }
```

HELPER.C

```
/*-------------------------------
   HELPER.C -- Helper routines
               (c) Paul Yao, 1996
  -------------------------------*/
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include "property.h"

//-------------------------------------------------------------
void SetButtonCheck (HWND hwndDlg, int CtrlID, BOOL bCheck)
     {
     HWND hwndCtrl = GetDlgItem (hwndDlg, CtrlID) ;
     if (bCheck)
          {
```

```
            Button_SetCheck (hwndCtrl, BST_CHECKED) ;
            }
     }

//------------------------------------------------------------------
BOOL QueryButtonCheck (HWND hwndDlg, int CtrlID)
     {
     HWND hwndCtrl = GetDlgItem (hwndDlg, CtrlID) ;
     int  nCheck = Button_GetCheck (hwndCtrl) ;
     return (nCheck == BST_CHECKED) ;
     }
```

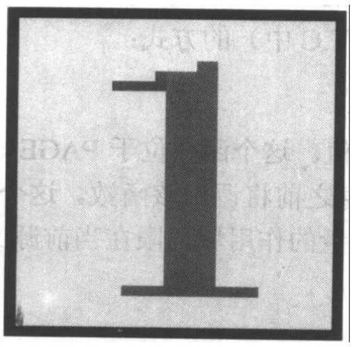
PROPERTY.RC

```
#include "property.h"
#include <windows.h>

IDI_APP                 ICON    DISCARDABLE     "PROPERTY.ICO"
IDI_PAGE1               ICON    DISCARDABLE     "page1.ico"
IDI_PAGE2               ICON    DISCARDABLE     "page2.ico"

MAIN MENU DISCARDABLE
     {
     POPUP "&Property-Sheet-Styles"
          {
          MENUITEM "PSH_WIZARD",                IDM_WIZARD
          MENUITEM SEPARATOR
          MENUITEM "PSH_HASHELP",               IDM_HASHELP
          MENUITEM "PSH_MODELESS",              IDM_MODELESS
          MENUITEM "PSH_NOAPPLYNOW",            IDM_NOAPPLYNOW
          MENUITEM "PSH_PROPTITLE",             IDM_PROPTITLE
          MENUITEM "PSH_RTLREADING",            IDM_RTLREADING
          }
     POPUP "&Window"
          {
          MENUITEM "Properties",                IDM_WINDOW_PROPERTIES
          }
     }

IDD_STYLES DIALOG DISCARDABLE  0, 0, 292, 127
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Window Styles"
FONT 8, "MS Sans Serif"
     {
     GROUPBOX        "&Initial State",IDC_STATIC,7,7,94,69
     CONTROL         "WS_VISIBLE",IDC_VISIBLE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,21,55,10
     CONTROL         "WS_DISABLED",IDC_DISABLED,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,35,64,10
     CONTROL         "WS_MINIMIZE",IDC_MINIMIZE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,49,60,10
     CONTROL         "WS_MAXIMIZE",IDC_MAXIMIZE,"Button",BS_AUTOCHECKBOX |
                     WS_TABSTOP,14,63,62,10
     GROUPBOX        "&Clipping",IDC_STATIC,7,80,94,42
     CONTROL         "WS_CLIPCHILDREN",IDC_CLIPCHILDREN,"Button",
                     BS_AUTOCHECKBOX | WS_TABSTOP,14,94,79,10
     CONTROL         "WS_CLIPSIBLINGS",IDC_CLIPSIBLINGS,"Button",
```

```
                BS_AUTOCHECKBOX | WS_TABSTOP,14,108,75,10
    GROUPBOX    "&Window Components",IDC_STATIC,115,7,170,98
    CONTROL     "WS_BORDER",IDC_BORDER,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,123,21,60,10
    CONTROL     "WS_CAPTION",IDC_CAPTION,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,123,35,60,10
    CONTROL     "WS_DLGFRAME",IDC_DLGFRAME,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,123,49,68,10
    CONTROL     "WS_HSCROLL",IDC_HSCROLL,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,123,63,71,10
    CONTROL     "WS_MAXIMIZEBOX",IDC_MAXIMIZEBOX,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,123,77,93,10
    CONTROL     "WS_MINIMIZEBOX",IDC_MINIMIZEBOX,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,123,91,90,10
    CONTROL     "WS_SYSMENU",IDC_SYSMENU,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,200,21,72,10
    CONTROL     "WS_THICKFRAME",IDC_THICKFRAME,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,200,35,77,10
    CONTROL     "WS_VSCROLL",IDC_VSCROLL,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,200,49,74,10
    }

IDD_EXSTYLES DIALOG DISCARDABLE  0, 0, 330, 161
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Extended Window Styles"
FONT 8, "MS Sans Serif"
    {
    GROUPBOX    "&Non-Client Area",IDC_STATIC,7,7,263,71
    CONTROL     "WS_EX_CLIENTEDGE",IDC_CLIENTEDGE,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,14,21,113,10
    CONTROL     "WS_EX_CONTEXTHELP",IDC_CONTEXTHELP,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,14,35,116,10
    CONTROL     "WS_EX_DLGMODALFRAME",IDC_DLGMODALFRAME,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,14,49,121,10
    CONTROL     "WS_EX_OVERLAPPEDWINDOW",IDC_EXOVERLAPPED,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,14,63,127,10
    CONTROL     "WS_EX_PALETTEWINDOW",IDC_PALETTE,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,149,21,113,10
    CONTROL     "WS_EX_STATICEDGE",IDC_STATICEDGE,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,149,35,102,10
    CONTROL     "WS_EX_TOOLWINDOW",IDC_TOOLWINDOW,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,149,49,101,10
    CONTROL     "WS_EX_WINDOWEDGE",IDC_WINDOWEDGE,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,149,63,102,10
    GROUPBOX    "&Shell Interface",IDC_STATIC,8,80,214,39
    CONTROL     "WS_EX_ACCEPTFILES",IDC_ACCEPTFILES,"Button",
                 BS_AUTOCHECKBOX | WS_TABSTOP,14,90,97,10
    CONTROL     "WS_EX_APPWINDOW",IDC_APPWINDOW,"Button",
                 BS_AUTOCHECKBOX | WS_TABSTOP,14,104,96,10
    CONTROL     "WS_EX_TOPMOST",IDC_TOPMOST,"Button",BS_AUTOCHECKBOX |
                WS_TABSTOP,118,90,88,10
    CONTROL     "WS_EX_TRANSPARENT",IDC_TRANSPARENT,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,118,104,102,10
    GROUPBOX    "&Child Window",IDC_STATIC,7,121,316,33
    CONTROL     "WS_EX_CONTROLPARENT",IDC_CONTROLPARENT,"Button",
                BS_AUTOCHECKBOX | WS_TABSTOP,14,135,105,10
    CONTROL     "WS_EX_MDICHILD",IDC_MDICHILD,"Button",BS_AUTOCHECKBOX |
```

```
                WS_TABSTOP, 124, 135, 77, 10
    CONTROL     "WS_EX_NOPARENTNOTIFY", IDC_NOPARENTNOTIFY, "Button",
                BS_AUTOCHECKBOX | WS_TABSTOP, 206, 135, 111, 10
    }
```

PAGE1.ICO



PAGE2.ICO

PROPERTY.ICO

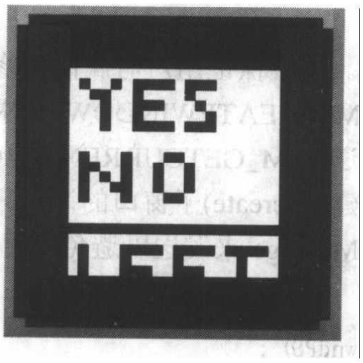


图 12-13　PROPERTY程序

为了响应 WM_CREATE 消息， PROPERTY 的主窗口创建了两个子窗口。标题为“ First Window ”的窗口是我们主要感兴趣的窗口。当其父窗口的大小发生改变时，这个窗口会相应地进行缩放以保证它占据其父窗口客户区的九分之一。当风格标志改变时，父窗口收到一条私有消息， PM_CREATEWINDOW ，响应这个消息，父窗口将清除现存的子窗口，并使用所要求的风格创建一个新的子窗口，然后给它自己发送一条 WM_SIZE 消息来强迫子窗口进行缩放。

创建子窗口的风格标志存储在两个全局变量中，该变量定义在 PROPERTY.C 中：

```
DWORD dwChildStyle = WS_CHILD | WS_VISIBLE | WS_BORDER | WS_CAPTION ;
DWORD dwChildExStyle = 0L ;
```

指向这些变量的指针被传递给 PROPSHEETPAGE 结构的 lParam 域中的每个属性页。例如，下面是将 dwChildStyle 传递给第一个属性页（在 SHEET.C 中）的方式：

```
pspage[0].lParam = (LPARAM) &dwChildStyle ;
```

第一个属性页在它的页过程(StylePageProc)中检索这个值。这个函数位于 PAGE1.C 中，在使用一个 PSPCB_CREATE 消息通知码创建属性页本身之前将调用该函数。这个指针被存储在一个被定义为 static 的全局变量中，因此该全局变量的作用域局限在当前源文件之内：

```
// 存储风格数据的指针
pTheStyles = (LPDWORD) ppsp->lParam;
```

其中 ppsp 是指向属性页结构 PROPSHEETPAGE 的指针，该指针将被传递给属性页函数。

每个属性页都有一个指针指向它负责编辑的数据，因此使改变生效的时候来到时，每个属性页对话框过程将改变的数据写回原来的数据中。这应该和使用全局变量——创建常规对话框时的通行做法——一样容易，但是我们用 lParam 参数来显示如何只使用 PROPSHEETPAGE 结构中的内容来初始化“远程”属性页，比如动态链接库中的一个属性页。

在属性页对话过程中，大多数令人感兴趣的事情都发生在对一条 WM_NOTIFY 消息的响应中。当按下 OK 或者 Apply 按钮时， PSN_APPLY 通知码被发送给对话过程。在前面已经提到过，属性页是以一种相当传统的方式创建的——即，它们是在用户激活单个页时才被创建。只有被创建了的属性页才会真正接到通知消息，这意味着当窗口风格属性页被关闭时，两个对话过程中可能只有一个能收到 PSN_APPLY 通知；或者，如果两个页都被创建了，它们将得到两个通知。

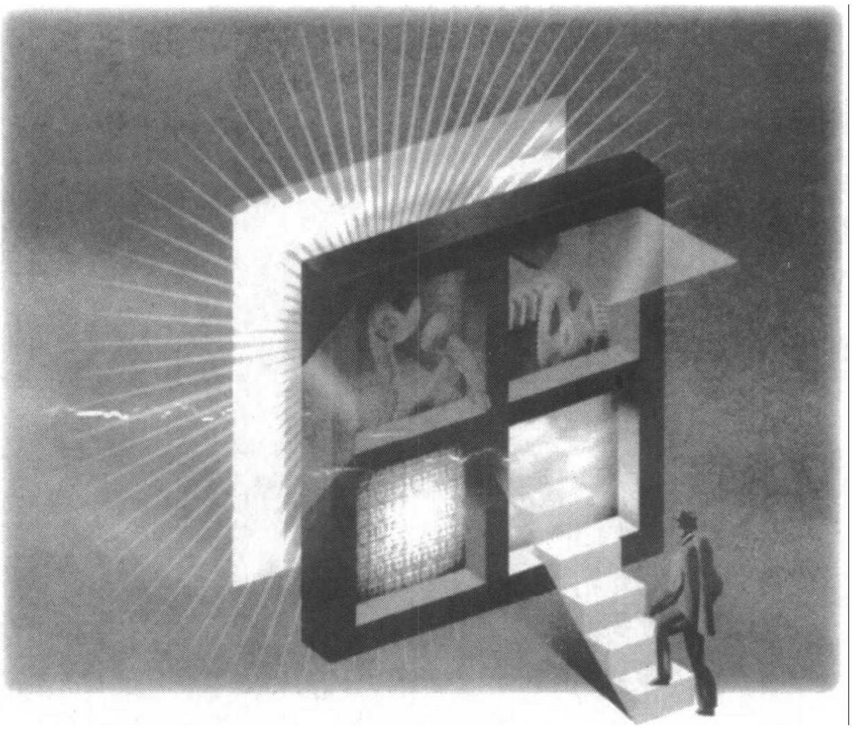
这就产生了收到两个 PSN_APPLY 通知的可能性。为了保证用户对属性的修改只被应用一此，每个对话过程应该在给程序的主窗口发送 PM_CREATEWINDOW 请求之前先检查它是否是活动的属性页。这可以通过给属性表格发送 PSM_GETCURRENTPAGEHWND 消息来做到，其结果是返回活动属性页的句柄。再创建(re-create)子窗口的请求只有从当前活动属性页的对话过程中才能发送出去——通过 PostMessage 来发送以避免定时问题：

```
HWND hwndPS = GetParent (hwndDlg) ;
HWND hwndActive = PropSheet_GetCurrentPageHwnd (hwndPS) ;

// 如果我们是活动页则只进行再创建(re-create)
if (hwndDlg == hwndActive)
    SendMessage (hwndMain, PM_CREATEWINDOW, 0, 0L) ;
```

PROPERTY 表明，你已经知道的大多数有关对话框的知识也可以用于属性表格。增加之处用来处理另外一些情形——创建向导、非模态属性表格以及使用一组不同的按钮等，这些东西有时对应用系统开发人员是很有用的。

第四部分

核心和打印机



第 13 章　存储管理与文件 I/O

如果你是在 Windows 95 中才首次学习 Windows 编程，那么你是一个非常幸运的家伙。你只是不知道你有多么的幸运，事实上，你将在本章中学习的课程可以用一句话来总结：当操纵内存和文件时，你几乎不需要使用除标准 C 运行库以外的任何东西。

我建议使用 C 库函数（诸如 malloc 、 free 、 fopen 、 fclose 等）的原因是这些函数很干净、很简单，你也许已经知道如何使用它们了，而且你也许非常习惯使用这些函数。但是最重要的是，在你的 Windows 95 程序中使用这些函数时不会出现什么问题。我们将在后面看到，情况并非总是这样的。

存储管理和文件输入/输出（ I/O ）是简单的非图形操作系统（如 MS_DOS ）提供给应用程序的两个最基本的服务；第三个是任务调度，非常简单，只是从磁盘文件将程序装入内存并启动该程序运行；在这三个服务之后，第四重要的服务也许就是跟踪日期和时间了。

由 Windows 95 内核提供的系统服务要广泛得多，包括动态链接（在第 19 章中讨论）、多任务、多线程和线程同步（第 14 章）、进程间通讯或者叫做 IPC（第 16 、 17 和 20 章）以及一些没有在这本书中涉及到的其它一些功能特征。

虽然使用 C 库函数很方便，但是也有可能编写一个根本没有使用任何 C 库函数的 Windows 95 程序。任何要求调用操作系统（如存储管理和文件 I/O 函数）的库函数都会有一个（并且通常有多个）相应的操作系统函数。你选择操作系统函数还是 C 库函数取决于你自己，并且可以混合或者分开使用它们。

13. 1　内存：好的、坏的和支离破碎的

为了说明在过去十年之内 Windows 的发展，我们来读一下 1985 年 11 月发布的 Windows 1.0 版本中的最低系统要求：“ 320K 内存、 DOS2.0 或者更高版本、两个双面磁盘驱动器、图形适配卡。”这既有趣又好笑。这些要求简单地反映了那时候大多数用户所使用的机器类型。从另一个角度来看， Microsoft 能让 Windows 在如此有限的环境中运行，这种成功是相当惊人的。 Windows 1.0 中的存储管理非常古怪，并且经常是十分令人讨厌，但它确实是可用的——至少在大多数时间内如此。

13. 1. 1　分段内存

Windows 1.0 是为 Intel 8086 和 8088 微处理器而设计的。从内部看来，它们是 16 位的

微处理器，有能力寻址 1MB 的内存。在 IBM 兼容机上，视频显示器和系统的基于 ROM 的 BIOS（基本输入/输出系统）保留了这个地址空间的高 384KB，为程序和数据剩下的最大 RAM 空间仅仅为 640KB。

寻址 1MB 内存要求 20 位的地址（2^{20} 是 1,048,576）。在 8086 和 8088 中，这个 20 位的物理地址是由两个 16 位的值，一个段地址和一个偏移地址形成的。微处理器有四个段地址：分别为代码段、数据段、堆栈段和附加段。一个 20 位的物理地址是通过将段地址左移四位，再加上偏移地址而形成的：

段地址：	ssssssssssssssss0000
+ **偏移地址：**	0000oooooooooooooooo
= **物理地址：**	aaaaaaaaaaaaaaaaaaaa

这样就得到了一个 20 位的地址，它可以存取多达 1MB 的内存（2 的 20 次幂）。

如果段寄存器保持为常数，那么程序只使用 16 位的偏移地址来寻址代码和数据。（由于 C 体系结构的影响，堆栈段被设置为与存储静态变量的数据段相同。）这两个段中的每一个都寻址 64KB 的内存。对于一个单任务操作系统，并且其中的程序只需要 64KB 代码和 64KB 数据，使用这种方式相当合理。

当应用系统程序越来越复杂（也越来越庞大）时，要求多段代码和/或数据。从而迫使 C 编译器的生产商定义近（或者短）指针，这种指针为 16 位，使用缺省的代码或者数据段，同时还定义远（或者长）指针，这种指针为 32 位，既包括一个偏移地址，又包括一个明确指定的段地址。然而， 32 位地址无法直接寻址内存，而且，这种 32 位的地址也不能没有处理偏移地址的回绕和对段地址进行相应调节的逻辑。同时， C 编译器的生产商还不得不定义不同的程序模式：小（一个代码段和一个数据段）、中（多个代码段）、紧缩（多个数据段）、大（多个代码段和数据段）和巨大（和大一样，但是具有内在的地址扩充逻辑）。

MD_DOS 本身对存储管理没有很多的支持，许多早期的 DOS 程序只是简单地指出有多少内存可用并将其用完。有胆量的程序员们为他们能够直接使用 PC 硬件而又不受操作系统烦琐之处的影响而骄傲。

因为 Windows 1.0 是一种多任务环境，因此有必要提供比 MS-DOS 更强的存储管理。考虑一下下面的情况：当多个程序被装入内存然后又退出，内存会变得支离破碎。操作系统必须对内存块进行移动以合并空闲空间。换句话说，即我在本书的前一个版本中所说的：“没有存储管理的多任务就和一个没有储藏室的晚会一样：你也许可以容纳一些早到的客人，但是一旦人们混到了一起，一些人的脚就会被踩脏。”

如何实现这一点呢？你不能仅仅移动内存中的一块代码或者一块数据，而不考虑应用程序，它将得到错误的地址。这就导致了分段内存的使用。如果程序只使用偏移地址，那么段地址可以被操作系统改变，而 Windows 的早期版本就是这么做的。

这种做法暗示了 Windows 程序被限制为小或者中模式，仅能有一个 64KB 的数据段。程序使用近指针来引用它们的数据段；某个进程的数据段地址在操作系统将控制交给程序时被设置。这允许 Windows 移动程序的数据段并重新设置段地址。程序所使用的所有远函

数调用（包括对操作系统函数的调用）在程序使用的代码段被装入内存时加以调整。

在 Windows 的 16 位版本上运行的程序既可以从它自己的数据段上分配内存（叫做“本地”内存，用 16 位的偏移量来引用），也可以在其数据段之外分配（叫做“全局”内存，通过 32 位的地址来引用）。在这两种情况下，内存分配函数都返回指向内存块的一个句柄。程序必须锁定该块以便在内存中固定它，锁定函数返回一个远指针。当使用完内存之后，必须解锁内存块，从而允许 Windows 在必要时移动该块。锁定和解锁的过程通常是在单个消息期间进行的，并且经常导致许多 bug，还带给程序员一些烦心事儿——比如头发过早脱落、消化不良乃至精神异常什么的。

和句柄的通常情况一样，内存句柄实际上也仅仅是指向 Windows 内核中一张表的指针。Windows 的早期版本保留了一个叫做 BURGERMASTER 的内存段，它包含一张句柄到内存的主转换表。这个名字来源于早期 Windows 开发人员所喜爱的一个餐厅，这个餐厅位于一条高速公路边，在 Washington 第一个 Microsoft 办公楼的对面（我曾经于几年前在 BurgetMaster 吃过饭，并得出一个结论：Windows 的开发人员一定都是跑步高手，因为他们总能越过高速公路而不出事故）。

除了历史背景外，讨论这个话题的另一个原因是 Windows 95 仍然保留了 Windows 1.0 中的一些寻址方案。当然，Windows 95 在体系结构上已经基本丢弃这种方案，只是它有时仍然会在函数调用的结构和语法中出现。

Windows 的 16 位版本中近地址和远地址所导致的问题也被带到了文件 I/O 中。在许多情况下，程序从对话框中接收到一个文件名，这是一个远地址。但是在小模式和中模式内存模块中，用于文件 I/O 的 C 库函数要求近地址。而且，将数据存储在全局内存块中的程序也必须使用远指针进行寻址，这样一来，涉及这些内存块的文件 I/O 就显得很不利索。这要求 Windows 重复 MS-DOS 汇编语言的文件 I/O 的函数调用。

13.1.2 中期的解决方案

我知道这种讨论已经引起了 MS-DOS 和 Windows 编程人员的痛苦回忆。另一个不受欢迎的回顾可能是 Lotus-Intel-Microsoft 的 LIM EMS 规范。这个规范通常用在大容量内存的机器上，这些内存通过位于未被视频适配卡和 ROM BIOS 占据的高端内存中的一个 64KB 的窗口，以 16KB 块的方式进行访问，16KB 内存的不同块可以被换入或者换出窗口。Windows 2.0 使用几个函数调用来支持 LIM EMS。

在 Windows 3.0 问世之前，Microsoft 就能够支持 Intel 286 的“保护模式”（相对于上面所讨论的 8086 和 8088 的“实模式”），而不对现存的程序造成太多的问题。在保护模式中，段地址被称作为一个“选择子”。它仍然是 16 位宽，但是在 286 内部，它引用一个 24 位的“基”地址，该基地址被加到偏移地址上，从而形成一个 24 位的可以寻址 16MB 内存的物理地址。

为了简化对 Windows 早期版本的转换，由全局内存分配函数返回的句柄只是选择子。保护模式的使用在一定程度上简化了存储管理，但是尚未真正达到目的。

选择子	ssssssssssssssss
	描述符表
基地址	bbbbbbbbbbbbbbbbbbbbbbbb
+偏移地址	00000000oooooooooooooooo
=物理地址	aaaaaaaaaaaaaaaaaaaaaaaa

13.1.3 最后是无可争议的 32 位

Windows 95 需要 Intel 386 、 486 或者奔腾微处理器，这些处理器使用 32 位寻址从而可以访问 2^{32}，即 4,294,967,296 字节（ 4GB ）的物理内存。当然，大多数 Windows 95 用户都不会达到这一极限。根据 Windows 95 的规模，操作系统只要求 4MB 内存，但是推荐使用 8MB 。目前，一般认为 16MB 为多数用途提供了足够的空间。

虽然 Intel 386 、 486 和奔腾微处理器仍然能够使用分段内存寻址，但是 Windows 95 保持段寄存器固定不变，而采用了一种 32 位的平坦（ flat ）寻址方案。这意味着， Windows 95 应用程序中的地址被简单地存储为 32 位值，并允许访问 4GB 内存。

由于 Windows 95 应用程序用于访问代码和数据的 32 位地址并非微处理器用来寻址物理内存的 32 位物理地址，应用系统使用的地址被称为“虚拟”地址，它通过一个“页表”被转换为物理地址。应用程序通常忽略这一过程（这是开发操作系统的原因之一）。你的程序似乎位于一个 32 位的地址空间中，你不需要进行任何特殊的操作来访问这一存储。然而，有些 Windows 技术文档谈到了虚拟地址和页表，因此理解其中的过程将是有所帮助的。

物理内存被分为长度是 4096 字节（ 4KB ）的“页”，因此，每页从最低 12 位为 0 的地址开始，具有 8MB 内存的机器有 2048 个页。 Windows 95 操作系统维护一个页表的集合（页表本身为 4KB 页），将虚拟地址转换为物理地址。

Windows 95 中的每个进程都有自己的“目录页”，这是连续存储的最多可达 1024 个的 32 位项目的集合。当前目录页的物理地址被存放在一个叫做 CR3 的微处理器寄存器中，当 Windows 95 在进程之间切换控制时，该地址将被改变。虚拟地址的头 10 位指定这个目录页中 1024 个可能项中的一个。在目录页中，项的头 20 位指定一个页表的物理地址（物理地址的低 12 位被设置为 0 ）。这就引用了另一页，该页也具有 1024 个 32 位的项。虚拟地址的中间 10 位引用这些项中的一个，该项再次用一个 20 位的物理地址来指示一个页帧的开始位置，此位置是一个物理地址。虚拟地址的低 12 位指向这个页帧中的一个物理位置。

我知道刚开始会有点迷惑。使用符号显示，你可以将一个 32 位的虚拟地址（即一个应用系统所使用的）表示为一个 10 位的目录页项（ d ）、一个 10 位的页表项（ p ）和一个 12 位的偏移（ o ）：

dddd-dddd-ddpp-pppp-pppp-oooo-oooo-oooo

对每一个进程，微处理器将一个 20 位的值存储在 CR3 寄存器中（r 代码寄存器）：

rrrr-rrrr-rrrr-rrrr-rrrr

过程的当前目录页的起始物理地址为：

rrrr-rrrr-rrrr-rrrr-rrrr-0000-0000-0000

记住所有的页都从 4KB 的整数倍的边界上开始，因此每页都开始于一个低 12 位等于 0 的地址。微处理器首先访问物理地址：

rrrr-rrrr-rrrr-rrrr-rrrr-dddd-dddd-dd00

这个位置包含另一个 20 位的值（t 代表表）：

tttt-tttt-tttt-tttt-tttt

这表示一个页表的起始物理地址：

tttt-tttt-tttt-tttt-tttt-0000-0000-0000

然后微处理器访问一个物理地址：

tttt-tttt-tttt-tttt-tttt-pppp-pppp-pp00

在这里存储的是一个页帧的 20 位值（f 代表帧）：

ffff-ffff-ffff-ffff-ffff

最后的 32 位物理地址是这个页帧和虚拟地址的低 12 偏移位的组合：

ffff-ffff-ffff-ffff-ffff-oooo-oooo-oooo

这就是物理地址了。

将虚拟地址转换为物理地址似乎是对时间的一种极大的浪费，但是实际上并非如此。Intel 386、486 和奔腾微处理器有一个内部的高速缓存，它可以将这些页表保存在处理器内。转换实际上进行得很快，而不会对性能造成多大的影响。在理论上，二级分页给每个应用系统提供了多达一百万个的 4KB 页。

分页的好处是巨大的：首先，应用系统相互隔离。没有一个进程会不小心覆盖了另一个进程的代码或数据空间，因为没有正确的 CR3 值，它甚至无法寻址到其它进程的内存，而 CR3 值的设置只能由 Windows 95 内核进行。

第二，分页技术解决了多任务环境中最基本的问题之一，即空闲内存的收集问题。在比较简单的寻址方案中，当多个程序执行并退出后，内存变得零零碎碎。如果内存变得过于零碎，那么程序将无法运行，因为它将没有足够的连续内存，即使空闲内存的总量是完全足够的。使用分页技术，则没有必要收集空闲的物理内存，因为页并不需要是连续的。每件事情都通过对页表的操作而得到处理。唯一的浪费实际上是来自页表本身和页的 4KB 粒度引起的开销。

第三，除了 20 位地址外，32 位页表项中有一些附加位。有一位指示某个页是否被访问过（这叫做“访问”位）；另一位指示该页是否被写入过（“脏”位）。Windows 95 可以使用这些位来确定内存的一页是否可以被交换到磁盘文件中，以获取更多的空闲物理内存。还有一位（“存在”位）指示该页是否被交换到磁盘上因而必须被再装入内存。

另有一个“读/写”位指示该页是否可以被写入，这一位保护代码避免由于指针的错误

而被写入。例如，如果你在一个 Windows 程序中包含下列语句：

```
*(int*)WinMain = 0;
```

你将得到一个消息框显示“这个程序执行了一个非法操作而将被关闭。”这一位并不禁止程序将源代码编译后在内存中存储要执行的汇编语言指令，我们将看到这一点。

关于 Windows 95 存储管理，还有几点注意事项：

虚拟地址为 32 位宽。你的程序代码和数据（静态的、堆栈的或者分配的）将具有 0x00000000 和 0x7FFFFFFF 之间的地址。Windows 95 本身使用 0x80000000 和 0xFFFFFFFF 之间的地址，而这正是你找到 Windows 95 动态链接库的入口点的地方。

空闲存储的总量是通过空闲物理内存和可用于页交换的空闲硬盘区域的总量来决定的。和通常的虚拟存储管理一样，Windows 95 使用一种最近最少使用（LRU）的算法来确定将哪些页交换到磁盘中。“访问”位和“脏”位在这里起作用。代码页不一定需要被保存到磁盘中去；因为代码页是不可写的，它们可以简单地从.EXE 文件或者动态链接库中被重新装入。

有时当你将鼠标从一个程序的客户区移动到另一个程序的客户区时，你将看到磁盘正在被访问。为什么会这样呢？Windows 必须将鼠标移动消息发送给另一个应用系统。如果程序中处理这个消息的代码当前不在内存中，那么 Windows 必须从磁盘文件中重新装入它。如果你有几个大型 Windows 程序同时被装入，而又没有太多的内存，那么当你在程序之间移动时，你将可能目睹一种“颠簸”（一种不规则的磁盘活动），因为此时 Windows 正在重新装入刚才被丢弃的页。当 Windows 在进行换入换出时，某些程序将突然慢下来（甚至完全停止）。

代码段可以在应用系统之间共享。对于动态链接库这尤其有用。同时运行的几个程序可以使用同样的 Windows 95 函数，而不需要将相同的代码多次装入内存中。代码的一个唯一拷贝是必要的。

当动态分配内存时，你分配的每一块并没有得到它自己的页。少量内存的连续分配将位于同一物理页，并校准到最近的 4 位开始地址（即，分配一个字节使用 16 个字节）。当某块被重新分配以更大的空间时，如果原来在页内紧跟在其后有其它内容，那么可能需要将它挪到内存中的其它地方。

除了页的 4KB 粒度外，物理内存不会变得支离破碎而不可挽救，因为碎片收集只涉及页表的处理。然而，如果一个应用系统分配、重分配并释放许许多多的内存块，那么应用系统的虚拟内存可能会变得零零碎碎。应用系统代码和数据的 2GB 极限通常足以避免这些问题。一个程序在达到虚拟存储的极限之前很可能已经用完了物理内存。但是超过极限的情况也可能发生，如果你的某个程序很可能会遇到这一问题，那么你也许想认真考虑一下“可移动的”内存，我将在本章后面加以讨论。

最后，在所有的这些开场白之后，我的建议仍然是一样的：尽量使用 C 的库函数。你有一个 32 位的地址空间，使用它甚至滥用它（如果你的应用系统真的很牛气的话）。

13.2 内 存 分 配

不知我是否强调够了使用 C 库函数分配内存？为了强化这种行为，我将从简短的复习开始。

13.2.1 C 的库函数

在你的程序中，可以定义一个类似如下所示的指针（例如，指向一个整数数组）：

```
int *p;
```

指针是一个未被初始化的 32 位值。你可以使用下面的语句获取一块由 p 指向的内存：

```
p= (int*) malloc (1024) ;
```

这个语句分配一块 1024 个字节的内存块，它可以存储 256 个 32 位整数。一个 NULL 指针表示内存分配没有成功。你也可以用下面的语句来分配这一内存块：

```
p = (int*) calloc (256, sizeof (int)) ;
```

传给 calloc 的两个参数相乘会得到 1024 个字节。 calloc 函数还具有将内存块初始化为 0 的作用。

如果你需要增加这一内存块的大小（也许增加一倍），那么可以调用：

```
p = (int*) realloc (p, 2048) ;
```

指针被传递给函数，从函数返回一个指针（可能是不同的值，尤其是在扩大内存块时）。这种语法说明操作系统（在这里即是 Windows 95 ）可能必须在虚拟内存中移动内存块。例如，如果你分配了一个 1024 字节的内存块，那么可能会得到一个 0x00750100 的虚拟地址。你也许还分配了第二个内存块，得到一个 0x00750504 的虚拟地址。不可能将第一个内存块增加 2048 字节而仍然使用同样的虚拟地址。在这种情况下， Windows 95 还必须将物理内存中的块移动到新的一页。

最后，当完成对内存的使用之后，你调用：

```
free (p) ;
```

这是由 ANSI C 规范定义的唯一的四个函数。C 编译器生产商通常还支持更多的函数，其中最有用的可能是_msize ，它返回一个已分配内存块的大小。

13.2.2 Windows 95 内存分配基础

我在前面已经提到过，你可以使用 C 库函数做的任何事都可以用你自己的代码或者通过使用 Windows 95 内核函数来做到。下面是用于为指向整数的指针分配一个内存块的 Windows 95 函数：

```
p = (int*)GlobalAlloc (uiFlags, dwSize) ;
```

函数要求两个参数：一系列可能的标志和被分配块的字节大小。它返回一个虚拟地址，你的程序可以使用该地址来寻址被分配的内存。返回 NULL 说明没有足够的内存可用于这一分配。

除了一个例外，对每个以单词 Global 开始的函数来说，相应地会有另一个以单词 Local 开始的函数。这两组函数在 Windows 95 中是相同的；两个不同的前缀是为了和 Windows 以前的版本兼容，在以前的版本中，Global 函数返回远指针，而 Local 函数返回近指针。

虽然定义得有点不同，但是 GlobalAlloc 的两个参数都是 32 位的无符号整数。如果将第一个参数设置为 0，那么你实际上使用了下面这个标志：

```
GMEM_FIXED (等于 0)
```

这使得 GlobalAlloc 在功能上等价于 malloc 函数。在 Windows 的早期版本中，GMEM_FIXED 标志是不鼓励的，因为 Windows 无法在物理内存中移动块。在 Windows 95 中，GMEM_FIXED 标志是正常的，因为它返回一个虚拟地址，而操作系统可以通过改变页表而在物理内存中移动内存块。

如果你希望将内存块的每个字节初始化为 0，那么可以使用标志：GMEM_ZEROINT。GPTR 标志将 GMEM_FIXED 和 GMEM_ZEROINT 标志组合在一起，如在 Windows 头文件中所定义的那样：

```
#define GPTR (GMEM_FIXED | GMEM_ZEROINT)
```

还有一个重分配函数：

```
p = (int*)GlobalReAlloc (p, dwSize, uiFlags) ;
```

如果是扩大内存块，那么你可以使用 GMEM_ZEROINT 标志将新的字节初始化为 0。

下面的函数返回内存块的大小：

```
dwSize = GlobalSize (p) ;
```

下面的函数释放内存块：

```
GlobalFree (p) ;
```

13.2.3 可移动的内存

我在前面已经提到过，在 Windows 以前的版本中，GMEM_FIXED 参数是不鼓励的，因为它要求 Windows 将内存块固定在物理内存；在 Windows 95 中，一个内存块可以在物理内存中被移动而保持相同的虚拟地址。即使这样，GlobalAlloc 函数仍然支持标志：

GMEM_MOVEABLE

以及如下的标识符来附带地将内存初始化为 0（在 Windows 头文件中定义）：

```
#define GHND (GMEM_MOVEABLE | GMEM_ZEROINT)
```

（注意多数字典使用拼写“movable”而不是“moveable”，这是为什么在本书的第一版本中，我的编辑器想要改变标识符的原因。不幸的是，Windows 的开发人员已经将它们的

拼写习惯刻在了石头上——至少刻在了 Windows 的头文件中了。）

GMEM_MOVEABLE 标志允许一个内存块在虚拟内存中被移动。这并非一定意味着内存块将在物理内存中也被移动，但是应用系统用于读写内存块的地址可以改变。这听起来似乎有点怪异？而实际上就是这样，我们马上就会看到这是如何做到的。

你也许会问：如果没有必要，为什么我要使用可移动内存呢（并且当你看到其中所涉及到的事情时，你也许会提出更犀利的问题）？一种回答是，为了保持和现存 Windows 源代码的兼容性。如果你的程序疯狂地分配、再分配和释放内存，那么虚拟内存当然会变得支离破碎。你的程序曾经在用完 4 或者 8 或者 16MB 的物理内存空间以前用完过 2GB 的虚拟内存吗？这有可能发生。问题随着人们连续运行其机器的趋势的增加而日益严重。连日运行的程序和那些为一两个小时运行而设计的程序相比，最终可能遇到更多的存储碎片。

如果这是一个潜在的问题，那么你将会希望使用可移动的内存，下面将告诉你如何做到这一点。首先，定义一个指针和一个类型为 GLOBALHANDLE 的变量：

```
int *p;
GLOBALHANDLE hGlobal ;
```

然后分配内存，例如：

```
hGlobal = GlobalAlloc (GHNE, 1024) ;
```

注意，由 GlobalAlloc 函数返回的值不需要进行强制类型转换。函数定义为返回一个 GLOBALHANDLE 变量，因为这是 GlobalAlloc 在 Windows 95 以前版本的一般情况下使用的方式。

和任何一种 Windows 句柄一样，不要过于担心数值的真实含义，只需将它存储起来。当你需要访问内存块时，调用：

```
p = (int*)GlobalLock (hGlobal) ;
```

从而将句柄转换为指针。在块被锁定期间， Windows 将固定住虚拟内存中的地址，它不能被移动。当你完成对块的访问时，调用：

```
GlobalUnlock (hGlobal) ;
```

从而使 Windows 95 可以自由地在虚拟内存中移动块。为了真正保证该过程的正确（并体会早期 Windows 程序员的痛苦），你应该在单个消息的处理过程中锁定和解锁内存块。

当你要释放内存时，使用句柄而不是指针调用 GlobalFree 。如果这时你没有该句柄，那么先如下获得它：

```
hGlobal = GlobalHandle (p) ;
```

你可以在解锁一个内存块之前多次锁定它。 Windows 维护一个锁定计数，每个锁定要求一个相应的解锁，然后才能释放内存块以进行移动。在虚拟内存中移动一个内存块并非真正地将字节从一个位置拷贝到另一个位置——它基本上只涉及对页表的操作。分配一个可移动块唯一真正的理由是防止虚拟内存的碎片化。

13.2.4 可丢弃内存

如果你勇敢到能使用 GMEM_MOVEABLE 选项，那么你也会勇敢到能使用：

```
GMEM_DISCARDABLE
```

这个选项只能和 GMEM_MOVEABLE 一起使用。使用这种标志分配的内存块可以在需要一些空闲空间时，由 Windows 内核从物理内存中丢弃。

听起来可能有点悬，但是仔细想一想，包含代码的内存块是可以丢弃的，这些块是不可写的，因此，从原始的.EXE 文件中装入代码将比把代码交换到磁盘再重装入快。如果为一块不可改变从而便于重新生成（通常由一个磁盘文件装入）的数据分配内存，那么你可以使它成为可丢弃的。当你调用 GlobalLock 得到一个 NULL 指针时，就知道它被丢弃了，然后可以重新生成数据。

一个内存块不能被丢弃，除非锁计数为 0 。为了有意地丢弃一个内存块，可以调用：

```
GlobalDiscard (hGlobal) ;
```

13.2.5 其它函数和标志

另一个在GlobalAlloc函数中可用的标志是GMEM_SHARE或GMEM_DDESHARE（它们是一样的）。正如它的名字所表明的，这个标志用于动态数据交换，我将在第 17 章中加以讨论。

GlobalAlloc 和 GlobalReAlloc 还可以包括 GMEM_NODISCARD 或者 GMEM_NOCOMPACT 标志，这些标志指示 Windows 不要为了满足内存的需要而丢弃任何的内存或者进行任何的内存移动。只有具有极端利他主义思想的程序员才会使用这些标志。

如果新指定的大小为零或者 GMEM_MODIFY 被包含在标志参数中，那么 GlobalReAlloc 函数还可以改变标志（例如将一个固定块改变为可移动的或者反之）。

GlobalFlags 函数返回 GMEM_DISCARDABLE 、 GMEM_DISCRARDED 和 GMEM_SHARE 标志的一个组合。

最后，你可以使用一个指向类型为 MEMORYSTATUS 结构的指针来调用 GlobalMemoryStatus ，从而确定物理内存的量和你的应用可用的虚拟内存的量。

这就完成了对以 Global 开始的函数的介绍。 Windows 95 还支持一些可以由你自己实现，也可以采用 C 库函数的函数。这些函数被称为 FillMemory （用某个字节填充）、ZeroMemory （用 0 填充）和 CopyMemory 与 MoveMemory ，后两个函数都是从一个位置拷贝内存到另一个位置。如果位置重叠，那么 CopyMemory 结构可能会不正确，可以使用 MoveMemory 取而代之。

13.2.6 指针合法吗

在你访问一些内存之前，可能希望测试它是否合法。如果指针非法，那么你的程序引

起一个保护异常。首先测试指针就保证了这种异常不会发生。IsBadCodePtr、IsBadReadPtr、IsBadWritePtr 和 IsBadStringPtr 函数可以为你做到这一点。其中第一个函数接受一个指针，如果指针非法，则返回 TRUE（非零）。其余三个以一个指针作为第一个参数，一个长度作为第二个参数。第四个函数还搜索一个 0 结束符，如果找到，则停止对内存合法性的测试。

13.2.7　虚拟内存函数

Windows 95 还支持一组以 Virtual 打头的函数，这些函数允许你对内存进行更多的控制。然而，只有相当独特的应用才会要求使用这些函数。

例如，假设你在编写一个集成的开发环境，包括编译器和执行器。在这一环境中，你的程序将读入源代码并将它编译到一个内存块。然后你可能想使该内存块仅供执行，以防止它不小心被刚编译好的可运行的程序读或者（更糟糕）写，这是你的开发环境试图做到的，以便更好地捕获用户程序中的错误。你也可以将某些内存块标志为“只读”。这些工作都可以使用虚拟内存函数来实现，这是很独特的要求。

另一种更普遍的需求是为数据保留一大块虚拟内存块，这些数据将随着你程序的运行而急剧扩展。如果只采取一般方法，比如经常使用的 C 的 realloc 函数或者 Windows 的 GlobalReAlloc 函数动态改变一个可分配块的大小，那么可能会引起性能问题。虚拟存储管理函数有助于避免对性能的影响。下面加以说明。

在 Windows 95 中，应用程序所用虚拟内存的任何部分都可能处于三种不同的状态之一——“占用”（即映射到一个物理内存区域）、“空闲”（可用于以后的内存分配）和“保留”，介于前两者之间。一块保留的虚拟内存没有对应任何物理内存，但是在保留内存块中的地址不会被使用，直到你希望它们被使用为止。这是你可以如何获得一大块连续虚拟内存的方法，在程序运行时，这块内存可能对应于物理内存，也可能没有。

为了使用虚拟内存函数，你的程序应该知道内存页的大小。和 Windows NT 不同，Windows 95 只运行在 Intel 微处理器上，并且页的大小总是 4096 个字节。如果你的程序被设计为还要在 Windows NT 上运行，那么使用 GetSystemInfo 函数来获取这个信息。这个函数要求一个参数，该参数是指向 SYSTEM_INFO 结构的指针。dwPageSize 域表示页的大小；lpMinimumApplicationAddress 和 lpMaximumApplicationAddress 域也有用，它们表示一个应用系统可用的最小和最大地址。对于 Windows 95，这些值分别是 0x00400000 和 0x7FFFFFFF。

VirtualAlloc 函数如下所示：

```
p = VirtualAlloc (pAddr, dwSize, iAllocType, iProtect) ;
```

第一个参数表示一个希望的虚拟内存起始基地址，当你首次调用这个函数时应该将它置为 NULL。第二个参数表示大小。第三个参数可以是 MEM_RESERVE 或者 MEM_COMMIT，分别表示保留或占用这个空间。第四个参数可以是以 PAGE_（如 PAGE_READONLY 或者 PAGE_EXECUTE）为前缀的常量，表示内存块的保护属性。以后对 VirtualAlloc 的调用可以占用或者保留这个块的一部分。VirtalFree 函数释放整个虚拟内存。

13.2.8 堆函数

最后一组内存分配函数以 Heap 打头。这些函数创建并维护一个连续的虚拟内存块，你可以从中分配更小的部分。你从调用 HeapCreate 函数开始，然后用 HeapAllocate 、HeapReAllocate 和 HeapFree 来分配和释放堆中的内存块。可以对堆进行紧缩来收集空闲的空间。

很少有什么理由需要用到堆函数。

13.3 文件输入/输出

我再说一次：对于文件的输入和输出，尽可能地使用 C 库函数。我已经在第 11 章中的 POPPAD3 程序中通过使用 fopen 、 fseek 、 fread 、 fwrite 和 fclose 这么做了。

13.3.1 以前的方式

在 Windows 下使用文件的方法在几年中正逐渐地加以改进。回到 Windows 1.0 和 2.0 的年代，唯一正规的文件 I/O 函数是 OpenFile ，而正式推荐用于读写文件的方法是编写小的汇编语言文件，直接访问 MS-DOS 的函数调用。虽然使用标准 C 运行库函数是可能的，但是在小模式或中模式的程序中，这些函数只通过近指针来使用，对于在全局内存块中保存文件数据的程序来说是很讨厌的，尽管文件名（通常是通过对话框获得的）是通过远指针访问的。

幸运的是，许多程序员立即发现了一些未公开的使用远指针访问文件的函数。这些函数使用的名字为_lopen 、 _lread 、 _lwrite 等等，并且它们几乎完全使用 MS-DOS 的功能调用。从 Windows 3.0 开始，这些函数被公开了，成为 Windows 编程的标准文件 I/O 函数。但是在 Windows 95 中不再推荐这些函数。

13.3.2 Windows 95 的不同之处

Windows 95 提供了对 Windows 早期版本中文件 I/O 的一些改进。

首先，和 Windows 3.1 类似， Windows 95 提供通用对话框库，其中包括 File Open 和 File Save 对话框，这些在第 11 章中介绍过。强烈推荐你使用这些对话框，它们避免了可能与系统有关的文件名分析。

第二， Windows 95 是一种 32 位环境。这意味着你可以使用单个的 fread 或者 fwrite 函数（或者相应的 Windows 等价函数）从文件中读入或者向文件中写入大量信息。对现有代码的改变涉及取消对 for 循环的使用。

第三， Windows 95 支持长文件名。你的程序处理长文件名的最简单方法是根本不介意它们。（够简单吧？） Windows 文档指示你应该使用由 GetVolumeInformation 返回的数

据动态地分配缓冲区来存储文件名，但是通常这会令人讨厌。我倾向于使用在 STDLIB.H 中定义的两个常量：_MAX_PATH （设置位 260 ）和_MAX_FNAME (256)静态地分配空间。

13.3.3 Windows 95 文件 I/O 函数

如果你不用 C 库函数进行文件 I/O ，那么可以使用由 Windows 95 提供的函数。CreateFile 函数的功能非常广泛：

```
hFile = CreateFile (szName, dwAccess, dwShare, NULL, dwCreate, dwFlag, 0) ;
```

尽管名字为 CreateFile ，但是该函数也用于打开现存的文件，而且，你使用相同的函数还可以打开管道（用于 IPC ）和通讯设备。

对文件来说，第一个参数为文件名。第二个参数可以是 GENERIC_READ 或者 GENERIC_WRITE ，或者二者的组合。使用 0 允许你获取关于文件的信息而不访问它。dwShare 参数可以打开可供共享的文件，允许其它进程读（ FILE_SHARE_READ ）或写（ FILE_SHARE_WRITE ）或者读写文件。

dwCreate 标志是几个常数之一，指示应该如何打开文件。这些标志是全面的，并且你完全可以顾名思义。 CREATE_NEW 标志在文件已经存在时将导致函数调用失败，而 CREATE_ALWAYS 将删除任何一个现存文件中的内容。类似地， OPEN_EXISTING 在文件不存在时将导致函数调用的失败，而 OPEN_ALWAYS 会在文件不存在时创建一个新的文件。 TRUNCATE_EXISTING 在文件不存在时将导致函数调用的失败，而在文件存在时，又将删除文件中的所有数据。

dwFlags 标志可以是以 FILE_ATTRIBUTE 和 FILE_FLAG 开始的常数的组合，用于设置文件属性和其它的行为特征。

CreateFile 返回一个类型为 HANDLE 的变量。最后你可以通过使用该句柄调用 CloseHandle 来关闭文件。 ReadFile 和 WriteFile 函数非常类似：

```
ReadFile (hFile, pBuffer, dwToRead &dwHaveRead, NULL) ;
WriteFile (hFile, pBuffer, dwToWrite, &dwHaveWritten, NULL) ;
```

第二个参数是一个指向包含数据的缓冲区的指针；第三个参数指示读出或者写入文件的字节个数；而第四个参数是一个指向由函数返回的变量的指针，它指示实际读出或者写入的字节个数（最后一个参数只用于使用 FILE_FLAG_OVERLAPPED 标志打开的文件，该话题不在本书讨论的范围之内）。

13.3.4 内存映射 I/O

在 Windows 95（这是对 Windows 的 16 位版本的一个巨大增强）中，当从一个文件中读出或者往一个文件中写入时，你可以将文件视为一块内存。刚开始，这可能有点不好理解，但是慢慢地会变得很方便。而且，对于在两个或者多个进程之间共享内存时，这种技术是值得推荐的，我们将在第 19 章中的“动态链接库”（ DLL ）中看到这一点。

这是进行文件映射 I/O 最简单的方法：首先，你按通常的方法使用 CreateFile 创建文件，然后调用：

```
hMap = CreateFileMapping (hFile, NULL, dwProtect, 0, 0, szName) ;
```

dwProtect 参数可以是下列值中的一个，并且应该和文件的共享模式相容：PAGE_READONLY、PAGE_WRITECOPY、PAGE_READWRITE。最后一个参数是一个可选的名字，通常用于和另一个进程共享数据。与此类似，OpenFileMapping 使用指定的名字打开同一个文件。两个函数都返回一个 HANDLE 数据类型。

当你需要访问整个文件中的一部分时，调用 MapViewOfFile:

```
p = MapViewOfFile (hMap, dwAccess, dwHigh, dwLow, dwNumber) ;
```

整个或者部分文件将被映射到内存中，内存开始于文件中由 dwHigh 或者 dwLow 指示的一个 64 位的偏移地址处(显然，dwHigh 将为 0，除非你有一个长度大于 4GB 的文件)。dwNumber 参数指示你希望映射到内存中的字节个数。dwAccess 参数可以是 FILE_MAP_WRITE（也允许读）或者是 FILE_MAP_READ，并且应该和 CreateFileMapping 调用中的 dwProtect 参数相容。

现在，你可以使用由函数返回的指针来访问或者修改文件中的数据。FlushViewOfFile 函数将内存映射文件中的所有脏页写回到磁盘中。UnmapViewOfFile 函数使 MapViewOfFile 返回的指针失效。你仍然必须使用 CloseHandle 来关闭文件。

我们将在关于动态链接库的那一章中看到这一过程的一个例子。

第 14 章 多任务和多线程

多任务是一个操作系统可以同时运行多个程序的能力。基本上，操作系统使用一个硬件时钟来为同时运行的每个进程分配“时间片”。如果时间片足够的小，并且机器也没有由于太多的程序而超负荷时，那么在用户看来，所有的这些程序似乎在同时运行着。

多任务并不是什么新的东西。在大型计算机上，多任务是必然的。这些大型机通常有几十甚至几百个终端和它连接，而每个终端用户都应该感觉到他或者她独占了整个计算机。另外，大型机的操作系统通常允许用户“后台提交工作”，这些后台作业可以在用户进行其它工作时，由机器动态完成。

个人计算机上的多任务花了更长的时间才变为现实。但是现在我们几乎进入了一个新纪元，PC 多任务也被认为是很正常的了。我马上就会讨论到，Windows 的 16 位版本在一定的程度上支持多任务，而现在我们有了 Windows NT 和 Windows 95，Windows 的这些 32 位版本支持真正的多任务，而且，作为一种额外的收获，它们还支持多线程。

多线程是在一个程序内部实现多任务的能力。程序可以把它自己分隔为单独的执行“线程”，这些线程似乎也同时在运行。这一概念初看起来似乎没有什么用处，但是它可以让程序使用多线程在后台执行长作业，从而使用户不必长时间地无法使用其计算机进行其它工作。（有时这也许不是所希望的，找个借口去冲冲凉或者到冰箱去看看总是很不错的！）

14.1 多任务的各种模式

在 PC 的早期，有人曾经提倡未来的多任务，但是大多数的人还是很迷惑：在一个单用户的个人计算机上，多任务有什么用呢？但是，最后事实表明多任务是并不知道这一概念的用户所需要的。

14.1.1 DOS 下的多任务

在最初 PC 上的 Intel 8088 微处理器并不是为多任务而设计的。部分原因（我在上一章中讨论过）是不够强的内存管理。当启动和结束多个程序时，多任务的操作系统通常需要移动内存块以收集空闲内存。在 8088 上是不可能透明于应用系统来做到这一点的。

DOS 本身对多任务没有太大的帮助，它的设计目标是小型，并且与应用无关，因此，除了装入程序以及给程序提供对文件系统的访问，它几乎没有提供任何支持。

不过，有创造性的程序员仍然在 DOS 的早期就找到了一种克服这些缺陷的方法，大多

数是使用终止并常驻（TSR）程序。有些 TSR，比如假脱机打印程序等，通过陷入硬件定时中断来执行真正的后台处理；其它的 TSR，诸如 SideKick 等弹出工具，可以执行某种类型的任务切换——挂起一个应用，运行弹出工具。DOS 也逐渐有所增强以便提供对 TSR 的支持。

一些软件生产商试图在 DOS 之上构造出任务切换或者多任务外壳（诸如 Quarterdeck 的 DesqView），但是在这些环境中，仅有一个占据了大量的市场，当然，这就是 Windows 了。

14.1.2 非抢占式的多任务

当 Microsoft 在 1985 年引入 Windows 1.0 时，它是最成熟的解决方案，目的是突破 DOS 的局限。Windows 在实模式下运行，但是即使这样，它已可以在物理内存中移动内存块，这是多任务的前提，虽然移动的方法尚未完全透明于应用，但是几乎可以忍受了。

在一种图形窗口环境中，多任务比在一种命令行单用户操作系统中显得更有意义。例如，在经典的命令行 UNIX 中，可以在命令行之外执行程序，让它们在后台运行，然而，程序的所有显示输出必须被重定向到一个文件中，否则输出将和用户正在做的事情混在一起。

窗口环境允许多个程序在相同屏幕上一起运行，前后切换非常容易，并且还可以快速地将数据从一个程序移动到另一个程序中，例如，将绘图程序中创建的图片嵌入由字处理程序编辑的文本文件中。在 Windows 中，以多种方式支持数据转移，首先是使用剪贴板，后来又使用动态数据交换（DDE），而现在则是通过对象链接和嵌入（OLE）。

不过，在早期 Windows 的多任务实现还不是多用户操作系统中传统的抢占式的时间分片。Windows 的这些 16 位版本支持一种被称为“非抢占式的多任务”，这种类型的多任务由于 Windows 基于消息的体系结构而成为可能。通常情况下，一个 Windows 程序将在内存中睡眠，直到它收到一条消息为止。这些消息通常是用户的键盘或鼠标输入的直接或间接结果。当处理完消息之后，程序将控制返回给 Windows。

Windows 的 16 位版本不会绝对地基于一个时间滴答将控制从一个 Windows 程序切换到另一个，任何的任务切换都发生在当程序完成对消息的处理后将控制返回给 Windows 时。这种非抢占式的多任务也被称为“合作式的多任务”，因为它要求来自应用方面的一些合作。一个 Windows 程序可以占用整个系统，如果它要花很长一段时间来处理消息的话。

虽然非抢占式的多任务是 16 位 Windows 的一般规则，但仍然出现了某些形式的抢占式多任务。Windows 使用抢占式多任务来运行 DOS 程序，而且，为了实现多媒体，还允许动态链接库接收硬件定时中断。

16 位 Windows 包括几个功能特性来帮助程序员解决——或者至少是对付——非抢占式多任务中的局限，最显著的当然是时钟式鼠标光标。当然，这并非一种解决方案，而仅仅是让用户知道一个程序正在忙于处理一件长作业，因而系统将在一段时间内不可使用。另一个部分的解决方案是 Windows 定时器，它允许程序周期性地接收消息并完成一些工作。定时器通常用于时钟应用和动画。

针对抢占式多任务的另一种解决方案是 PeekMessage 函数调用，我们曾在第 4 章中的

RANDRECT 程序里看到过。一个程序通常使用 GetMessage 调用从它的消息队列中检索下一条消息，不过，如果在消息队列中没有消息，那么 GetMessage 不会返回，一直到出现一条消息为止。而另一方面， PeekMessage 将控制返回程序，即使没有等待的消息。这样，一个程序可以执行一个长作业，并在代码中混入 PeekMessage 调用。只要没有这个程序或其它任何程序的挂起消息，那么这个长作业将继续运行。

14. 1. 3　PM 和串行化的消息队列

Microsoft 在一种半 DOS/半 Windows 的环境下实现多任务的第一个尝试(和 IBM 合作）是 OS/2 和显示管理器（ PM ）。虽然 OS/2 明确地支持抢占式多任务，但是这种抢占式似乎并未在显示管理器中得以实现。问题在于 PM 串行化来自键盘和鼠标的用户输入消息。这意味着，直到前一个用户输入消息被完全处理以后， PM 不会将一个键盘或者鼠标消息传送给程序。

尽管键盘和鼠标消息只是一个 PM （或者 Windows ）程序可以接收的许多消息中的几个，大多数的其它消息都是键盘或者鼠标事件的结果。例如，菜单命令消息是用户使用键盘或者鼠标进行菜单选择的结果。在处理菜单命令消息时，键盘或者鼠标消息并未完全被处理。

串行化消息队列的主要原因是允许用户的预“键入”和预“按入”。如果一个键盘或者鼠标消息导致输入焦点从一个窗口切换到另一个窗口，那么接下来的键盘消息应该进入拥有新的输入焦点的窗口中去。因此，系统不知道将下一个用户输入消息发送到何处，直到前一条消息被处理完以后。

目前的共识是不应该让一个应用系统有可能占用整个系统，而这要求一个非串行化的消息队列， Windows NT 和 Windows 95 支持这种消息队列。如果一个程序正在忙于处理一项长作业，那么你可以将输入焦点切换到另一个程序中。

14. 1. 4　多线程解决方案

我讨论 OS/2 的显示管理器，只是因为它是第一个为早期的 Windows 程序员（比如我自己）介绍多线程的环境。有趣的是， PM 实现多线程的局限为程序员提供了应该如何构造多线程程序的必要线索。即使这些局限在现在的 Windows 95 中已经有了大量的减少，但是从更有限的环境中学到的经验仍然是非常有效的。因此，让我们继续。

在一种多线程环境中，程序可以将它们自己分隔为同时运行的片（叫做执行的线程）。对线程的支持是解决 PM 中存在的串行化消息队列的最好方法，并且在 Windows 95 中线程有更实际的意义。

就代码来说，一个线程简单地被表示为可能调用程序中其它函数的函数。程序从其主线程开始执行，这个主线程在传统的 C 程序中叫做 main 的函数，而在 Windows 中是 WinMain 。一旦运行起来，程序可以通过在 system 调用中指定初始线程函数的名字来创建新的线程的执行。操作系统在线程之间抢占式地切换控制，和它在进程之间切换控制的方法非常类似。

在 OS/2 的显示管理器中，每个线程可以创建一个消息队列，也可以不创建。如果希望从线程创建窗口，那么一个 PM 线程必须创建消息队列。否则，如果只是进行许多的数据处理或者图形输出，那么线程不需要创建消息队列。因为无消息队列的进程不处理消息，所以它们将不会挂起系统。唯一的限制是一个无消息队列线程无法向一个消息队列线程中的窗口发送消息，或者调用任何发送消息的函数（不过，它们可以将消息 post 给消息队列线程）。

这样，PM 程序员学会了如何将它们的程序分隔为一个消息队列线程（在其中创建所有的窗口并处理传送给窗口的消息）和一个或者多个无消息队列线程，在其中执行长的后台任务。PM 程序员还了解到“1/10 秒规则”，大体上，程序员被告知，一个消息队列线程处理任何消息都不应该超过 1/10 秒，任何花费更长时间的事情都应该在另一个线程中完成。如果所有的程序员都遵循这一规则，那么将没有 PM 程序会将系统挂起超过 1/10 秒。

14.1.5 多线程体系结构

我已经说过 PM 的局限为程序员理解如何在运行于图形环境中的程序里使用多个线程的执行提供了必要的线索。因此在这里我将为你的程序建议一种体系结构：你的主线程创建你程序所需要的所有窗口，并在其中包括所有的窗口过程，以便处理这些窗口的所有消息；其它的所有线程只进行一些后台处理，除了和主线程通讯，它们不和用户进行交流。

考虑这种体系结构的一种方法是：主线程处理用户输入（和其它消息），并创建进程中的其它线程，这些附加的线程完成与用户无关的任务。

换句话说，你程序的主线程是一个总头目，而你的其它线程是总头目的职员。总头目将大的工作分配给他或者她的职员，而他或者她保持和外部世界的联系。因为它们仅仅是职员，所以其它线程不会举行它们自己的记者招待会。它们会认真地完成自己的工作，将结果报告给总头目，并等待他们的下一个任务。

一个程序中的线程是同一进程的不同部分，因此他们共享进程的资源，如内存和打开的文件。因为线程共享程序的内存，所以他们还共享静态变量。然而，每个线程都有他们自己的堆栈，因此动态变量对每个线程是唯一的。每个线程还有它自己的处理器状态（和数学协处理器状态），这个状态在进行线程切换期间被保存和恢复。

14.1.6 线程间的“争吵”

正确地设计、编程和调试一个复杂的多线程应用显然是 Windows 程序员可能遇到的最困难的工作之一。因为抢占式多任务系统可以在任何时刻中断一个线程，并将控制切换到另一个线程中，在两个线程之间可能有无法预料的随机交互的情况。

多线程程序中的一个常见的错误被称为“竞争条件”，这发生在程序员假设一个线程在另一个线程需要某数据之前已经完成了某些处理（如准备数据）的时候。为了帮助协调线程的活动，操作系统要求各种形式的同步。一种是信号量，它允许程序员在代码中的某一点阻止一个线程的执行，直到另一个线程发信号让它继续为止。类似于信号量的是“临界区”，它是代码中不可中断的部分。

但是信号量还可能引入称为“死锁”的常见线程相关的错误，这发生在两个线程互相阻止了另一个的执行，而继续执行的唯一办法又是它们继续向前运行。

幸运的是，32 位程序比 16 位程序更能抵抗线程所涉及的某些问题。例如，假定一个线程执行下面的简单语句：

```
lCount++ ;
```

其中 lCount 是由其它线程使用的一个 32 位的 long 型变量，C 中的这个语句被编译为两个机器代码指令，第一个将变量的低 16 位加 1，而第二个指令将任何可能的进位加到高 16 位上。假定操作系统在这两个机器代码指令之间中断了线程。如果 lCount 在第一个机器代码指令之前是 0x0000FFFF，那么 lCount 在线程被中断时为 0，而这正是另一个线程将看到的值。只有当线程继续执行时，lCount 才会增加到正确的值 0x00010000。

这是那些偶尔会导致操作问题的错误之一。在 16 位程序中，解决此问题正确的方法是将语句包含在一个临界区中，在这期间线程不会被中断。然而，在一个 32 位程序中，该语句是正确的，因为它被编译为一条机器代码指令。

14.1.7 Windows 的好处

Windows NT 和 Windows 95 有一个所希望的消息队列。这种实现似乎非常好：如果一个程序正在花费一段长时间处理一条消息，那么鼠标位于该程序的窗口上时，鼠标光标将呈现为一个时钟，但是当将鼠标移到另一个程序的窗口上时，鼠标光标将变为正常的箭头形状。只需按一下就可以将另一个窗口提到前台。

然而，用户仍然不能使用正在完成大作业的那个程序，因为大作业会阻止程序接收其它消息，这不是所希望的。一个程序应该总是对消息开放的，所以需要使用从属线程。

在 Windows NT 和 Windows 95 中，消息队列线程和无消息队列线程之间没有什么区别，每个线程在创建时都会有它自己的消息队列，从而减少了 PM 程序中关于线程的一些不便的规则。（然而，在大多数情况下，你仍然想通过一条线程的消息过程处理输入，而将长作业传递给那些不包含窗口的线程，这种结构几乎总是最容易理解的，我们将看到这一点。）

还有更好的事情：Windows NT 和 Windows 95 中有个函数允许线程杀死同一进程中的另一个线程。当你开始编写多线程代码时，你将会发现这种功能在有时是很方便的。OS/2 的早期版本没有“杀死线程”的函数。

最后的好消息（至少对这里的话题是好消息）是 Windows NT 和 Windows 95 实现了一些被称为“线程局部存储”（TLS）的功能。为了理解这一点，回顾一下我在前面提到过的，静态变量（对一个函数来说，既是全局又是局部变量）在线程之间是被共享的，因为它们位于进程的数据存储空间中。动态变量（对一个函数来说总是局部变量）对每一个线程则是唯一的，因为它们占据堆栈上的空间，而每个线程都有它自己的堆栈。

有时让两个或多个线程使用相同的函数，而让这些线程使用唯一于线程的静态变量是很方便的。这就是线程局部存储，其中涉及一些 Windows 函数调用，但是 Microsoft 还为 C 编译器进行扩展，使线程局部存储的使用更透明于程序员。

14.1.8 新的！先进的！有线程了！

既然已经介绍了线程的现状，让我们来展望一下线程的未来。有时，存在一种去使用操作系统所提供的每一种功能特性的趋势。最坏的情况是，当你的老板走到你的桌前并说，“我听说这种新的某某东西非常火。让我们在自己的程序中用一些这个某某东西吧。”然后你将花费一个星期的时间，试图去了解这个某某东西可以如何帮助你的应用。

应该注意的是，在并不需要多线程的应用系统中加入多线程是没有任何意义的。如果你的程序显示时钟光标的时间太长，或者如果它使用 PeekMessage 调用来避免时钟光标，那么请重构你的程序，使用多线程可能会是一个好主意。否则，你是在为难你自己，并可能会在代码中引入新的错误。

在某些情况下，时钟光标可能是完全适当的。我在前面提到过“ 1/10 秒规则”，而将一个大文件装入内存可能会花费多于 1/10 秒的时间，这是否意味着文件装入例程应该在分离的线程中实现呢？没有必要。当用户命令一个程序打开文件时，他或者她通常想立即完成该操作。将文件装入例程放在分离的线程中只会增加额外的负担。完全不值得这样做，即使你想向你的朋友夸耀你在编写多线程程序！

14.2 Windows 95 多线程

让我们立刻来看一些使用多线程的简单程序。

14.2.1 再论随机矩形

图 14-1 中的 RNDRCTMT 程序是第 4 章里的 RANDRECT 程序的多线程版本，你将回忆起 RANDRECT 使用的是 PeekMessage 循环来显示一系列的随机矩形。

RNDRCTMT.MAK

```
#------------------------
# RNDRCTMT.MAK make file
#------------------------

rndrctmt.exe : rndrctmt.obj
     $(LINKER) $(GUIFLAGS) -OUT:rndrctmt.exe rndrctmt.obj $(GUILIBS)

rndrctmt.obj : rndrctmt.c
     $(CC) $(CFLAGSMT) rndrctmt.c
```

RNDRCTMT.C

```
/*-----------------------------------------
   RNDRCTMT.C -- Displays Random Rectangles
                 (c) Charles Petzold, 1996
  -----------------------------------------*/
```

```
#include <windows.h>
#include <process.h>
#include <stdlib.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HWND hwnd ;
int  cxClient, cyClient ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "RndRctMT" ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Random Rectangles",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     return msg.wParam ;
     }

VOID Thread (PVOID pvoid)
     {
     HBRUSH hBrush ;
     HDC    hdc ;
```

```
     int    xLeft, xRight, yTop, yBottom, iRed, iGreen, iBlue ;

     while (TRUE)
          {
          if (cxClient != 0 || cyClient != 0)
               {
               xLeft   = rand () % cxClient ;
               xRight  = rand () % cxClient ;
               yTop    = rand () % cyClient ;
               yBottom = rand () % cyClient ;
               iRed    = rand () & 255 ;
               iGreen  = rand () & 255 ;
               iBlue   = rand () & 255 ;

               hdc = GetDC (hwnd) ;
               hBrush = CreateSolidBrush (RGB (iRed, iGreen, iBlue)) ;
               SelectObject (hdc, hBrush) ;

               Rectangle (hdc, min (xLeft, xRight), min (yTop, yBottom),
                               max (xLeft, xRight), max (yTop, yBottom)) ;

               ReleaseDC (hwnd, hdc) ;
               DeleteObject (hBrush) ;
               }
          }
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_CREATE :
               _beginthread (Thread, 0, NULL) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 14-1 RNDRCTMT程序

首先请注意，在 make 文件中有一些小的改变。在编译步骤中， CFLAGS 变量被改变为 CFLAGSMT 。除了包含-MT 标志外，这个变量和 CFLAS 是相同的，-MT 标志是编译器在编译多线程应用时所需要的。具体说，编译器将在.OBJ 文件中插入 LIBCMT.LIB 文件名，而不是 LIBC.LIB ，连接器将使用该信息。

LIBC.LIB 和 LIBCMT.LIB 文件包含 C 库函数，有些 C 库函数包含静态数据。例如，

由于 strtok 函数可能被连续地多次调用，所以它在静态内存中保存了一个指针；在多线程程序中，每个线程必须在 strtok 函数中有它自己的静态指针。因此，这个函数的多线程版本稍微不同于单线程的 strtok 函数。

同时请注意我在 RNDRCTMT.C 中包含了头文件 PROCESS.H ，这个文件定义一个名为_beginthread 的函数，它启动一个新的线程（我们马上将看到）。只有定义了_MT 标识符，才会定义这个文件，这是-MT 标志的另一个结果。

在 WinMain 函数中，由 CreateWindow 返回的 hwnd 值被保存在一个全局变量中，因此 cxClient 和 cyClient 值也可以由窗口过程的 WM_SIZE 消息获得。

窗口过程以最容易的方法调用_beginthread——简单地以线程函数的地址（称为 Thread ）作为第一个参数，其它参数使用 0 ，线程函数返回 VOID 并有一个参数是一个指向 VOID 的指针。在 RNDRCTMT 中的 Thread 函数不使用这个参数。

在调用了_beginthread 函数之后，线程函数（以及该线程函数可能调用的其它任何函数）中的代码和程序中的其它代码同时运行。两个或者多个线程使用一个进程中的同一函数，在这种情况下，动态局部变量（存储在堆栈上）对每个线程是唯一的；对进程中的所有线程来说，所有的静态变量都是一样的。这就是窗口过程设置全局的 cxClient 和 cyClient 变量并由 Thread 函数使用的方式。

有时你需要唯一于各个线程的持久的数据。通常，这种数据是静态变量，但在 Windows 95 中，你可以使用“线程局部存储”，我将在本章后面进行讨论。

14.2.2　程序设计竞赛的问题

1986 年 10 月 3 日， Microsoft 举行了为期一天的针对计算机出版社的技术编辑和作者的简短的记者招待会，来讨论他们当时的一组语言产品，包括他们的第一个交互开发环境，QuickBASIC 2.0 。当时， Windows 1.0 出现还不到一年，但是没有人知道我们什么时候能得到与该环境类似的东西。（这用了好几年。）这一事件与众不同的部分原因是由 Microsoft 的公关人员所炮制的“ Storm the Gates ”程序设计竞赛。 Bill Gates 使用 QuickBasic 2.0 ，而计算机出版社的人员可以使用他们选择的任何语言产品。

竞赛的问题是从公众所提出的题目中挑选出来的（挑选那些需要大约半小时编程的问题），问题如下：

创建一个包含四个窗口的多任务模拟。第一个窗口必须显示一系列的递增数，第二个必须显示一系列的递增素数，而第三个必须显示 Fibonacci 序列（ Fibonacci 序列以数字 0 和 1 开始，每下一个数是其前两个数的和——即 0 、 1 、 1 、 2 、 3 、 5 、 8 等等）。这三个窗口应该在数字达到窗口底部时或者进行滚动，或者自己清除自己。第四个窗口必须显示任意半径的圆，而程序必须在按下一个 Escape 键时终止。

当然，在 1986 年 10 月，在 DOS 下运行的这样一个程序最多只能是多任务模拟，而且没有一个竞赛者足够勇敢——并且其中大多数也没有足够的知识——来为 Windows 编写这个程序。并且，如果真要这么做，当然不会只花半小时了！

参加这次竞赛的大多数人编写了一个程序来将屏幕分为四个区域，程序中包含一个循环，依次更新每个窗口，然后检查是否按下了 Escape 键。作为 DOS 环境下的传统习惯，

程序占用了百分之百的 CPU 处理。

如果在 Windows 1.0 中编程，那么结果将是类似图 14-2 所示的程序。我说“类似”，是因为我编写的是 32 位程序，但是程序结构和相当多的代码——除了变量和函数参数定义——都是相同的。

MULTI1.MAK

```
#----------------------
# MULTI1.MAK make file
#----------------------

multi1.exe : multi1.obj
     $(LINKER) $(GUIFLAGS) -OUT:multi1.exe multi1.obj $(GUILIBS)

multi1.obj : multi1.c
     $(CC) $(CFLAGS) multi1.c
```

MULTI1.C

```
/*---------------------------------------
   MULTI1.C -- Multitasking Demo
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int cyChar ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "Multi1" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
```

```
     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Multitasking Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;
     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     return msg.wParam ;
     }

int CheckBottom (HWND hwnd, int cyClient, int iLine)
     {
     if (iLine * cyChar + cyChar > cyClient)
          {
          InvalidateRect (hwnd, NULL, TRUE) ;
          UpdateWindow (hwnd) ;
          iLine = 0 ;
          }
     return iLine ;
     }

// Window 1: Display increasing sequence of numbers
// ----------------------------------------------

LRESULT APIENTRY WndProc1 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum, iLine ;
     static short cyClient ;
     char         szBuffer[16] ;
     HDC          hdc ;

     switch (iMsg)
          {
          case WM_SIZE :
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_TIMER :
               if (iNum < 0)
                    iNum = 0 ;

               iLine = CheckBottom (hwnd, cyClient, iLine) ;

               wsprintf (szBuffer, "%d", iNum++) ;
```

```
               hdc = GetDC (hwnd) ;
               TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
               ReleaseDC (hwnd, hdc) ;

               iLine++ ;

               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Window 2: Display increasing sequence of prime numbers
// ------------------------------------------------------

LRESULT APIENTRY WndProc2 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum = 1, iLine ;
     static short cyClient ;
     char         szBuffer[16] ;
     int          i, iSqrt ;
     HDC          hdc ;

     switch (iMsg)
          {
          case WM_SIZE :
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_TIMER :
               do   {
                    if (++iNum < 0)
                         iNum = 0 ;

                    iSqrt = (int) sqrt (iNum) ;

                    for (i = 2 ; i <= iSqrt ; i++)
                         if (iNum % i == 0)
                              break ;
                    }
               while (i <= iSqrt) ;

               iLine = CheckBottom (hwnd, cyClient, iLine) ;

               wsprintf (szBuffer, "%d", iNum) ;

               hdc = GetDC (hwnd) ;
               TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
               ReleaseDC (hwnd, hdc) ;

               iLine++ ;

               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

```
// Window 3: Display increasing sequence of Fibonacci numbers
// ----------------------------------------------------------

LRESULT APIENTRY WndProc3 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int   iNum = 0, iNext = 1, iLine ;
     static short cyClient ;
     char         szBuffer[16] ;
     int          iTemp ;
     HDC          hdc ;

     switch (iMsg)
          {
          case WM_SIZE :
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_TIMER :
               if (iNum < 0)
                    {
                    iNum  = 0 ;
                    iNext = 1 ;
                    }

               iLine = CheckBottom (hwnd, cyClient, iLine) ;

               wsprintf (szBuffer, "%d", iNum) ;

               hdc = GetDC (hwnd) ;
               TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
               ReleaseDC (hwnd, hdc) ;

               iTemp  = iNum ;
               iNum   = iNext ;
               iNext += iTemp ;

               iLine++ ;

               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Window 4: Display circles of random radii
// ----------------------------------------

LRESULT APIENTRY WndProc4 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static short cxClient, cyClient ;
     HDC          hdc ;
     int          iDiameter ;

     switch (iMsg)
          {
          case WM_SIZE :
```

```
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_TIMER :
               InvalidateRect (hwnd, NULL, TRUE) ;
               UpdateWindow (hwnd) ;

               iDiameter = rand() % (max (1, min (cxClient, cyClient))) ;

               hdc = GetDC (hwnd) ;

               Ellipse (hdc, (cxClient - iDiameter) / 2,
                             (cyClient - iDiameter) / 2,
                             (cxClient + iDiameter) / 2,
                             (cyClient + iDiameter) / 2) ;

               ReleaseDC (hwnd, hdc) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Main window to create child windows
// -----------------------------------

LRESULT APIENTRY WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char    *szChildClass[] = { "Child1", "Child2",
                                        "Child3", "Child4" } ;
     static HWND    hwndChild[4] ;
     static WNDPROC ChildProc[] = { WndProc1, WndProc2,
                                    WndProc3, WndProc4 } ;
     HINSTANCE      hInstance ;
     int            i, cxClient, cyClient ;
     WNDCLASSEX     wndclass ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

               wndclass.cbSize        = sizeof (wndclass) ;
               wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
               wndclass.cbClsExtra    = 0 ;
               wndclass.cbWndExtra    = 0 ;
               wndclass.hInstance     = hInstance ;
               wndclass.hIcon         = NULL ;
               wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
               wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
               wndclass.lpszMenuName  = NULL ;
               wndclass.hIconSm       = NULL ;

               for (i = 0 ; i < 4 ; i++)
                    {
```

```
                    wndclass.lpfnWndProc    = ChildProc[i] ;
                    wndclass.lpszClassName = szChildClass[i] ;

                    RegisterClassEx (&wndclass) ;

                    hwndChild[i] = CreateWindow (szChildClass[i], NULL,
                         WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                         0, 0, 0, 0, hwnd, (HMENU) i, hInstance, NULL) ;
                    }

               cyChar = HIWORD (GetDialogBaseUnits ()) ;
               SetTimer (hwnd, 1, 10, NULL) ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               for (i = 0 ; i < 4 ; i++)
                    MoveWindow (hwndChild[i], (i % 2) * cxClient / 2,
                                              (i > 1) * cyClient / 2,
                                cxClient / 2, cyClient / 2, TRUE) ;
               return 0 ;

          case WM_TIMER :
               for (i = 0 ; i < 4 ; i++)
                    SendMessage (hwndChild[i], WM_TIMER, wParam, lParam) ;

               return 0 ;

          case WM_CHAR :
               if (wParam == '\x1B')
                    DestroyWindow (hwnd) ;

               return 0 ;

          case WM_DESTROY :
               KillTimer (hwnd, 1) ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 14-2 MULTI1程序

在这个程序里实际上没有什么我们没见过的东西。主窗口创建四个子窗口，每个子窗口占据客户区的一个象限。主窗口还设置一个 Windows 定时器并发送 WM_TIMER 消息给四个子窗口中的每一个。

通常一个 Windows 程序应该保留足够的信息以便在 WM_PAINT 消息期间重建其窗口中的内容。MULTL1 没有这么做，既然它绘制和清除窗口的速度如此之快，所以我认为那是不必要的。

素数产生器的效率并不很高，但是有效。如果一个数除了 1 和它自身以外没有别的因

数，那么这个数就是素数。当然，要检查一个数是否是素数并不要求使用小于被检查数的所有数来除这个数并检查余数，而只需使用所有小于被检查数的平方根的数。平方根计算是引入浮点数的原因，否则，该程序将是完全基于整数的程序。

MULTI1 程序没有什么不好的地方。使用 Windows 定时器是在 Windows 的早期版本和 Windows 95 中模拟多任务的一种好方法，然而，定时器的使用有时限制了程序的速度。如果程序可以在 WM_TIMER 消息处理中更新它的所有窗口而还有时间富余下来的话，那就意味着它并没有充分利用我们的机器资源。

一种可能的解决方案是在单个 WM_TIMER 消息期间进行两次或者更多次的更新，但是到底多少次呢？这将不得不依赖于机器的速度，而这有很大的可变性。你当然不会想编写一个程序只能适应 25Mhz 的 386，或者只能适应 50Mhz 的 486 或 500Mhz 的 786。

14.2.3 多线程解决方案

让我们来看一看关于这个程序设计问题的一种多线程解决方案。MULTI2 程序如图 14-3 所示。

MULTI2.MAK

```
#----------------------
# MULTI2.MAK make file
#----------------------

multi2.exe : multi2.obj
     $(LINKER) $(GUIFLAGS) -OUT:multi2.exe multi2.obj $(GUILIBS)

multi2.obj : multi2.c
     $(CC) $(CFLAGSMT) multi2.c
```

MULTI2.C

```
/*---------------------------------------
   MULTI2.C -- Multitasking Demo
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <process.h>

typedef struct
     {
     HWND hwnd ;
     int  cxClient ;
     int  cyClient ;
     int  cyChar ;
     BOOL bKill ;
     }
```

```
     PARAMS, *PPARAMS ;

LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Multi2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Multitasking Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

int CheckBottom (HWND hwnd, int cyClient, int cyChar, int iLine)
     {
     if (iLine * cyChar + cyChar > cyClient)
          {
          InvalidateRect (hwnd, NULL, TRUE) ;
          UpdateWindow (hwnd) ;
          iLine = 0 ;
          }
     return iLine ;
     }
```

```
// Window 1: Display increasing sequence of numbers
// ----------------------------------------------

void Thread1 (PVOID pvoid)
     {
     int     iNum = 0, iLine = 0 ;
     char    szBuffer[16] ;
     HDC     hdc ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     while (!pparams->bKill)
          {
          if (iNum < 0)
               iNum = 0 ;

          iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                               pparams->cyChar, iLine) ;

          wsprintf (szBuffer, "%d", iNum++) ;

          hdc = GetDC (pparams->hwnd) ;

          TextOut (hdc, 0, iLine * pparams->cyChar,
                   szBuffer, strlen (szBuffer)) ;

          ReleaseDC (pparams->hwnd, hdc) ;

          iLine++ ;
          }
     _endthread () ;
     }

LRESULT APIENTRY WndProc1 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params ;

     switch (iMsg)
          {
          case WM_CREATE :
               params.hwnd = hwnd ;
               params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
               _beginthread (Thread1, 0, &params) ;
               return 0 ;

          case WM_SIZE :
               params.cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_DESTROY :
               params.bKill = TRUE ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
```

```
     }

// Window 2: Display increasing sequence of prime numbers
// ------------------------------------------------------

void Thread2 (PVOID pvoid)
     {
     char    szBuffer[16] ;
     int     iNum = 1, iLine = 0, i, iSqrt ;
     HDC     hdc ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     while (!pparams->bKill)
          {
          do
               {
               if (++iNum < 0)
                    iNum = 0 ;

               iSqrt = (int) sqrt (iNum) ;

               for (i = 2 ; i <= iSqrt ; i++)
                    if (iNum % i == 0)
                         break ;
               }
          while (i <= iSqrt) ;

          iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                               pparams->cyChar, iLine) ;

          wsprintf (szBuffer, "%d", iNum) ;

          hdc = GetDC (pparams->hwnd) ;

          TextOut (hdc, 0, iLine * pparams->cyChar,
                   szBuffer, strlen (szBuffer)) ;

          ReleaseDC (pparams->hwnd, hdc) ;

          iLine++ ;
          }
     _endthread () ;
     }

LRESULT APIENTRY WndProc2 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params ;

     switch (iMsg)
          {
          case WM_CREATE :
               params.hwnd = hwnd ;
               params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
```

```
                _beginthread (Thread2, 0, &params) ;
                return 0 ;

           case WM_SIZE :
                params.cyClient = HIWORD (lParam) ;
                return 0 ;

           case WM_DESTROY :
                params.bKill = TRUE ;
                return 0 ;
           }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Window 3: Display increasing sequence of Fibonacci numbers
// ----------------------------------------------------------

void Thread3 (PVOID pvoid)
     {
     char     szBuffer[16] ;
     int      iNum = 0, iNext = 1, iLine = 0, iTemp ;
     HDC      hdc ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     while (!pparams->bKill)
          {
          if (iNum < 0)
               {
               iNum  = 0 ;
               iNext = 1 ;
               }

          iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                               pparams->cyChar, iLine) ;

          wsprintf (szBuffer, "%d", iNum) ;

          hdc = GetDC (pparams->hwnd) ;

          TextOut (hdc, 0, iLine * pparams->cyChar,
                   szBuffer, strlen (szBuffer)) ;

          ReleaseDC (pparams->hwnd, hdc) ;

          iTemp  = iNum ;
          iNum   = iNext ;
          iNext += iTemp ;

          iLine++ ;
          }
     _endthread () ;
     }
```

```
LRESULT APIENTRY WndProc3 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static PARAMS params ;

     switch (iMsg)
          {
          case WM_CREATE :
               params.hwnd = hwnd ;
               params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
               _beginthread (Thread3, 0, &params) ;
               return 0 ;

          case WM_SIZE :
               params.cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_DESTROY :
               params.bKill = TRUE ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Window 4: Display circles of random radii
// -----------------------------------------

void Thread4 (PVOID pvoid)
     {
     HDC     hdc ;
     int     iDiameter ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     while (!pparams->bKill)
          {
          InvalidateRect (pparams->hwnd, NULL, TRUE) ;
          UpdateWindow (pparams->hwnd) ;

          iDiameter = rand() % (max (1,
                           min (pparams->cxClient, pparams->cyClient))) ;

          hdc = GetDC (pparams->hwnd) ;

          Ellipse (hdc, (pparams->cxClient - iDiameter) / 2,
                        (pparams->cyClient - iDiameter) / 2,
                        (pparams->cxClient + iDiameter) / 2,
                        (pparams->cyClient + iDiameter) / 2) ;

          ReleaseDC (pparams->hwnd, hdc) ;
          }
     _endthread () ;
     }

LRESULT APIENTRY WndProc4 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
```

```
     {
     static PARAMS params ;

     switch (iMsg)
          {
          case WM_CREATE :
               params.hwnd = hwnd ;
               params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
               _beginthread (Thread4, 0, &params) ;
               return 0 ;

          case WM_SIZE :
               params.cxClient = LOWORD (lParam) ;
               params.cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_DESTROY :
               params.bKill = TRUE ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

// Main window to create child windows
// ----------------------------------

LRESULT APIENTRY WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char   *szChildClass[] = { "Child1", "Child2",
                                       "Child3", "Child4" } ;
     static HWND    hwndChild[4] ;
     static WNDPROC ChildProc[] = { WndProc1, WndProc2,
                                    WndProc3, WndProc4 } ;
     HINSTANCE      hInstance ;
     int            i, cxClient, cyClient ;
     WNDCLASSEX     wndclass ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

               wndclass.cbSize        = sizeof (wndclass) ;
               wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
               wndclass.cbClsExtra    = 0 ;
               wndclass.cbWndExtra    = 0 ;
               wndclass.hInstance     = hInstance ;
               wndclass.hIcon         = NULL ;
               wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
               wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
               wndclass.lpszMenuName  = NULL ;
               wndclass.hIconSm       = NULL ;

               for (i = 0 ; i < 4 ; i++)
                    {
```

```
                    wndclass.lpfnWndProc    = ChildProc[i] ;
                    wndclass.lpszClassName = szChildClass[i] ;

                    RegisterClassEx (&wndclass) ;

                    hwndChild[i] = CreateWindow (szChildClass[i], NULL,
                         WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                         0, 0, 0, 0, hwnd, (HMENU) i, hInstance, NULL) ;
                    }

               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;

               for (i = 0 ; i < 4 ; i++)
                    MoveWindow (hwndChild[i], (i % 2) * cxClient / 2,
                                              (i > 1) * cyClient / 2,
                                  cxClient / 2, cyClient / 2, TRUE) ;
               return 0 ;

          case WM_CHAR :
               if (wParam == '\x1B')
                    DestroyWindow (hwnd) ;

               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 14-3 MULTI2程序

MULTI2.C 的 WinMain 和 WndProc 函数非常类似于 MULTI1.C 中的同名函数。WndProc 为四个窗口注册了四种窗口类，创建了这些窗口，并在 WM_SIZE 消息期间缩放这些窗口。WndProc 的唯一不同是它不再设置 Windows 定时器，也不再处理 WM_TIMER 消息。

MULTI2 中较大的改变是每个子窗口过程通过在 WM_CREATE 消息期间调用 _beginthread 函数来创建另一个线程的执行。总的来说，MULTI2 程序有五个同时运行的线程的执行，主线程包含主窗口过程和四个子窗口过程，其余的四个线程使用名为 Thread1、Thread2 等的函数，这四个线程负责绘制四个窗口。

我在 RNDRCTMT 程序中给出的多线程代码没有使用_beginthread 的第三个参数，这个参数允许一个创建另一个线程的线程在 32 位变量中将信息传递给其他线程。通常，这个变量是一个指针，而且是指向一个结构的指针，这允许创建线程和新线程共享信息，而不必借助于全局变量。你可以看到，在 MULTI2 中没有全局变量。

对 MULTI2 程序，我在程序头定义了一个名为 PARAMS 的结构和一个名为 PPARAMS

的指向结构的指针，这个结构有五个域——窗口句柄、窗口的宽度和高度、字符的高度和名为 bKill 的布尔变量。最后的结构域允许创建进程告知被创建进程何时终止。

让我们来看一看 WndProc1 ，这是显示递增数序列的子窗口过程。窗口过程变得非常简单，唯一的局部变量是一个 PARAMS 结构。在 WM_CREATE 消息期间，它设置这个结构的 hwnd 和 cyChar 域，并调用_beginthread 来创建一个使用 Thread1 函数的新线程，并传递给新线程一个指向该结构的指针。在 WM_SIZE 消息期间， WndProc1 设置结构的 cyClient 域，而在 WM_DESTROY 消息期间，它将 bKill 域设置为 TRUE 。 Thread1 函数通过对_endthread 的调用而告结束。这并不是绝对必要的，因为线程将在退出线程函数之后被清除。不过，当退出一个深陷入某种复杂的处理层次的线程时，_endthread 是有用的。

Thread1 函数完成在窗口上的实际绘图，并且和程序的其它四个线程同时运行。函数接收指向 PARAMS 结构的一个指针，并进入一个 while 循环，每次检查 bKill 是 TRUE 还是 FALSE 。如果是 FALSE ，那么函数必须进行 MULTI1.C 中的 WM_TIMER 消息期间所作的同样处理——格式化数字、获取设备环境句柄并使用 TextOut 显示数字。

当你在 Windows 95 中运行 MULTI2 时，将会看到，窗口更新要比在 MULTI1 中快得多，这表明程序在更加有效地利用处理器的资源。在 MULTI1 和 MULTI2 之间还有另一种区别：通常，当你移动或者缩放一个窗口时，缺省窗口过程进入一种模态循环，而窗口的所有输出都将停止；在 MULTI2 中，输出将继续。

14.2.4 有问题吗

似乎 MULT2 程序并没有达到它应该有的坚固性。我为什么会这样认为呢？让我们来看一看 MULTI2.C 中的一些多线程“缺陷”，以 WndProc1 和 Thread1 为例。

WndProc1 在 MULTI2 的主线程中运行，而 Thread1 与它同时运行， Windows 95 在这两个线程之间进行切换是不可预测的。假定 Thread1 正在运行，并且刚好执行了检查 PARAMS 结构的 bKill 域是否为真的代码。发现不为真，但是这之后 Windows 95 将控制切换到主线程，这时用户终止了程序， WndProc1 收到一个 WM_DESTROY 消息并将 bKill 参数置为 TRUE 。哦，太晚了！操作系统突然切换到 Thread1 中，而该函数会试图获取一个不存在的窗口的设备环境句柄。

事实证明，这不是一个问题。 Windows 95 足够地牢靠，以致图形函数只是失败而已，不会引起任何问题。

正确的多线程编程技术涉及线程同步的使用（尤其是临界区的使用），我将马上加以详细地讨论。大体上，临界区通过对 EnterCriticalSection 和 LeaveCriticalSection 的调用而加以界定。如果一个线程进入一个临界区，那么另一个线程将无法再进入这个临界区。后一个线程被阻止在对 EnterCriticalSection 的调用上，直到第一个线程调用 LeaveCriticalSection 。

在 MULTI2 中的另一个可能存在的问题是，当另外一个线程显示其输出时，主线程可能会收到一条 WM_ERASEBKGND 或者 WM_PAINT 消息。这里，使用临界区有助于避免当两个进程试图在同一个窗口上绘图时可能导致的任何问题。但是，经验表明， Windows 95 很恰当地串行化了对图形绘制函数的访问，即，当另一个线程正在绘图的时候，一个线

程不能在同一个窗口上绘图。

Windows 95 文档提醒说，有一种未进行图形函数串行化的情形，这就是 GDI 对象（如画笔、画刷、字体、位图、区域和调色板等）的使用。有可能发生一个线程清除了一个对象，而另一个线程仍然在使用它的情况。解决这个问题的方法要求使用临界区，或者最好不要在线程之间共享 GDI 对象。

14.2.5 Sleep 的好处

我曾经提到，我认为对一个多线程程序来说，最好的体系结构是主线程创建程序中的所有窗口，以及所有的窗口过程，并处理所有的窗口消息。其它线程完成后台工作或者长作业。

不过，假设你想在另一个线程中做动画。通常，Windows 中的动画是使用 WM_TIMER 消息来实现的。如果这个线程没有创建窗口，那么它也不会收到这些消息；如果没有定时器，动画又可能会运行得太快。

解决方案是 Sleep 函数。实际上，线程调用 Sleep 函数来自愿地挂起自己，仅有的一个参数是以毫秒计的时间。Sleep 函数调用不会返回，直到指定的时间过去为止。在这段时间内，线程被挂起，并不会被分配给时间片（尽管该线程显然仍然要求在时钟滴答时给予一小段的处理时间，因为系统必须确定线程是否应该重新开始执行）。给 Sleep 一个值为 0 的参数将导致线程交回它尚未使用完的时间片。

当一个线程调用 Sleep 时，只是该线程被挂起指定的时间；系统仍然运行其它的线程，这些线程和挂起线程可以是在同一个进程中，也可以是在另一个进程中。我在第 4 章中的 SCRAMBLE 程序中使用了 Sleep 函数，以放慢擦除操作。

通常，你不应该在你的主线程中使用 Sleep 函数，因为这会减慢对消息的处理速度，但是因为 CRAMBLE 没有创建任何窗口，因此在那里使用 Sleep 应该没有问题。

14.3 线程同步

大约每年一次，在我宿舍窗口外面交通繁忙地段的红绿灯会停止工作。结果是造成交通的混乱，虽然轿车一般能避免撞上别的轿车，但是这些车经常挤在一起。

我把两条路相交的十字路口用术语称为“临界区”。一辆向南的车和一辆向西的车不可能同时通过一个十字路口而不撞着对方。依赖于交通流量，可以采用不同的方法来解决这个问题。对于视野清楚车辆稀少的路口，可以相信司机有处理的能力。车辆增多可能会要求一个停车标志，而更加繁忙的交通则将要求有红绿灯，红绿灯有助于协调路口的交通（当然，需要这些灯工作正常）。

14.3.1 临界区

在单任务操作系统中，传统的计算机程序不需要红绿灯来帮助协调它们之间的行为。

它们在运行时似乎独占了整条路，而且也确实是这样，没有什么会干扰它们的工作。

即使在多任务操作系统中，大多数的程序也似乎各自独立地在运行，但是可能会发生一些问题。例如，两个程序可能会需要同时从同一个文件中读或者对同一文件进行写。在这种情况下，操作系统提供了一种共享文件和记录上锁的技术来帮助解决这个问题。

然而，在支持多线程的操作系统中，情况会变得混乱而且存在潜在的危险。两个或多个线程共享某些数据的情况并不罕见，例如，一个线程可以更新一个或者多个变量，而另一个线程可以使用这些变量。有时这会引发一个问题，而有时又不会（记住操作系统将控制从一个线程切换到另一个线程的操作，只能在机器代码指令之间发生。如果只是一个整数被线程共享，那么对这个变量的改变通常发生在单个指令中，因此潜在的问题被最小化了）。

然而，假设线程共享几个变量或者数据结构。通常，这么多个变量或者结构的域在它们之间必须是一致的。操作系统可以在更新这些变量的过程中间中断一个线程，那么使用这些变量的线程得到的将是不一致的数据。

结果是冲突的发生，并且通常不难想象这样的错误将对程序造成怎样的破坏。我们所需要的是类似于红绿灯的编程技术，以帮助我们对线程交通进行协调和同步，这就是临界区。大体上，一个临界区就是一块不可中断的代码。

有四个函数用于临界区。要使用这些函数，你必须定义一个临界区对象，这是一个类型为 CRITICAL_SECTION 的全局变量。例如：

```
CRITICAL_SECTION cs ;
```

这个 CRITICAL_SECTION 数据类型是一个结构，但是其中的域只能由 Windows 内部使用。这个临界区对象必须首先被程序中的某个线程初始化，通过调用：

```
InitializeCriticalSection (&cs) ;
```

这样就创建了一个名为 cs 的临界区对象。该函数的联机帮助包括下面的警告：“临界区对象不能被移动或者拷贝，进程也不能修改该对象，但必须在逻辑上把它视为不透明的。”这句话可以被解释为：“不要干扰它，甚至不要看它。”

当临界区对象被初始化之后，线程可以通过下面的调用进入临界区：

```
EnterCriticalSection (&cs) ;
```

在这时，线程被认为“拥有”临界区对象。没有两个线程可以同时拥有临界区对象，因此，如果一个线程进入了临界区，那么下一个使用同一临界区对象调用 EnterCriticalSection 的线程将在函数调用中被挂起。只有当第一个线程通过下面的调用离开临界区时，函数才会返回：

```
LeaveCriticalSection (&cs) ;
```

这时，第二个线程——在其 EnterCriticalSection 调用中被挂起的那个——将拥有临界区，其函数调用也将返回。

当临界区不再被程序所需要时，可以通过调用：

```
DeleteCriticalSection (&cs) ;
```

将其删除，该函数释放所有被分配来维护此临界区对象的系统资源。

这种临界区技术涉及“互斥”（此术语在我们继续讨论线程同步时将再次出现）。在任何时刻，只有一个线程能拥有一个临界区。因此，一个线程可以进入一个临界区，设置一个结构的域，然后退出临界区。另一个使用该结构的线程在访问结构中的域之前也要先进入该临界区，然后再退出临界区。

注意，你可以定义多个临界区对象，比如 cs1 和 cs2 。例如，如果一个程序有四个线程，而前两个线程共享一些数据，那么它们可以使用一个临界区对象，而另外两个线程共享一些其它的数据，那么它们可以使用另一个临界区对象。

你在主线程中使用临界区时应该小心。如果从属线程在它自己的临界区中花费了一段很长的时间，那么它可能会将主线程阻止一段不算短的时间。

14.3.2　互斥(mutex)对象

临界区的一个局限是它们只能用于在同一进程内的线程之间的协调。但是在某些情况下，你需要协调两个不同进程对同一资源的共享（如共享内存等）。在这里你无法使用临界区，但是可以使用一种被称为“互斥对象”的技术。

单词“ mutex ”是人为构造的，代表“ mutual exclusion （互斥）”，它在这里精确地表达了我们的目的。我们想防止当一个程序的线程正在更新数据或者使用更新过的数据时被中断，或者，使用我前面的那个比喻，我们想使南行的车辆和西行的车辆避免相撞。

14.4　事 件 信 号

我们可以将一个“大作业”定义为一个可能会违反 1/10 秒规则的程序。显然大作业包括字处理器中的拼写检查、数据库程序中的文件排序或者索引、电子表格的重计算、打印，甚至包括复杂的绘图。当然，迄今为止我们知道，遵循 1/10 秒规则的最好方法是将大作业分配到另一个线程的执行中。这些额外的线程不会创建窗口，因此它们不受 1/10 秒规则的限制。

通常希望这些额外的线程在完成其任务时能够通知主线程，或者主线程能够停止其它线程正在进行的作业。这就是我们下面将要讨论的。

14.4.1　BIGJOB1 程序

作为一个假想的大作业，我将使用一系列浮点运算，有时这种运算被称为“野蛮的（ savage ）”性能测试指标。这种计算以一种间接的方式增加一个整数：它求一个数的平方，再对结果取平方根（得到原来的整数），然后使用 log 和 exp 函数（同样得到原来的

整数），接着使用 atan 和 tan 函数（还是得到原来的整数），最后对结果加 1 。

BIGJOB1 函数如图 14-4 所示。

BIGJOB1.MAK

```
#-----------------------
# BIGJOB1.MAK make file
#-----------------------

bigjob1.exe : bigjob1.obj
     $(LINKER) $(GUIFLAGS) -OUT:bigjob1.exe bigjob1.obj $(GUILIBS)

bigjob1.obj : bigjob1.c
     $(CC) $(CFLAGSMT) bigjob1.c
```

BIGJOB1.C

```
/*---------------------------------------
   BIGJOB1.C -- Multithreading Demo
                (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <math.h>
#include <process.h>

#define REP              100000

#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE     (WM_USER + 0)
#define WM_CALC_ABORTED  (WM_USER + 1)

typedef struct
     {
     HWND hwnd ;
     BOOL bContinue ;
     }
     PARAMS, *PPARAMS ;

LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BigJob1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
```

```
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Multithreading Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void Thread (PVOID pvoid)
     {
     double  A = 1.0 ;
     INT     i ;
     LONG    lTime ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     lTime = GetCurrentTime () ;

     for (i = 0 ; i < REP && pparams->bContinue ; i++)
          A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;

     if (i == REP)
          {
          lTime = GetCurrentTime () - lTime ;
          SendMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
          }
     else
          SendMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;

     _endthread () ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
```

```
{
static char  *szMessage[] = { "Ready (left mouse button begins)",
                              "Working (right mouse button aborts)",
                              "%d repetitions in %ld msec" } ;
static INT    iStatus ;
static LONG   lTime ;
static PARAMS params ;
char          szBuffer[64] ;
HDC           hdc ;
PAINTSTRUCT   ps ;
RECT          rect ;

switch (iMsg)
     {
     case WM_LBUTTONDOWN :
          if (iStatus == STATUS_WORKING)
               {
               MessageBeep (0) ;
               return 0 ;
               }

          iStatus = STATUS_WORKING ;

          params.hwnd = hwnd ;
          params.bContinue = TRUE ;

          _beginthread (Thread, 0, &params) ;

          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_RBUTTONDOWN :
          params.bContinue = FALSE ;
          return 0 ;

     case WM_CALC_DONE :
          lTime = lParam ;
          iStatus = STATUS_DONE ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_CALC_ABORTED :
          iStatus = STATUS_READY ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;

          wsprintf (szBuffer, szMessage[iStatus], REP, lTime) ;

          DrawText (hdc, szBuffer, -1, &rect,
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
```

```
               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 14-4　BIGJOB1程序

这是一个相当简单的程序，但是我认为你将看到它如何演示在多线程程序中完成大作业的通用方法。为了使用 BIGJOB1 程序，在窗口的客户区中按下鼠标左键，从而开始野蛮的性能测试计算的 100,000 次重复，这在一台 33Mhz 的 386 机器上将花费 8 秒。当完成计算时，花费的时间将显示在窗口上。当正在进行计算时，你可以通过在客户区中按下鼠标右键来终止它。

让我们来看一看这是如何实现的：

窗口过程保持了一个被叫做 iStatus 的静态变量（该变量可以被设置为在程序开始处定义的三个常数之一，常数以 STATUS 为前缀），该变量表明程序是否准备好进行一次计算，是否正在进行一次计算，或者是否完成了计算。程序在 WM_PAINT 消息期间使用 iStatus 变量在客户区的中央显示一个适当的字符串。

窗口过程还保持一个静态结构（类型为 PARAMS ，也定义在程序的顶部），该结构是在窗口过程和其它线程之间的共享数据。结构只有两个域—— hwnd （程序窗口的句柄）和 bContinue ，这是一个布尔变量，用于指示线程是否继续计算或者停止。

当你在客户区中按下鼠标左键时，窗口过程将 iStatus 变量置为 STATUS_WORKING ，并设置 PARAMS 结构中的两个域。结构的 hwnd 域被设置为窗口句柄，当然， bContinue 被设置为 TRUE 。

然后窗口过程使用对_beginthread 的调用创建另一个线程。这个线程的函数叫做 Thread ，它以调用 GetCurrentTime 开始， GetCurrentTime 获取以毫秒计的 Windows 启动以来已经运行了的时间。然后它进入一个 for 循环，重复 100,000 次的野蛮测试计算。还要注意，如果 bContinue 被置为了 FALSE ，那么线程将退出循环。

在 for 循环之后，线程函数检查它是否确实完成了 100,000 次计算。如果是，那么它再次调用 GetCurrentTime 获得所经过的时间，然后使用 SendMessage 向窗口过程发送一个由程序定义的 WM_CALC_DONE 消息，经过的时间作为 lParam 参数。如果计算是在未完成之前被终止的（即，如果在循环期间 PARAMS 结构的 bContinue 域变为了 FALSE ），那么线程将发送给窗口过程一个 WM_CALC_ABORTED 消息。然后，线程通过调用_endthread 正常地结束。

在窗口过程中，当你在客户区中按下鼠标右键时， PARAMS 结构的 bContinue 域被置为 FALSE 。这是如何在完成计算之前结束计算的方法。

窗口过程处理 WM_CALC_DONE 消息时，首先保存经过的时间。对 WM_CALC_DOEN 和 WM_CALC_ABORTED 消息的处理都是通过对 InvalidateRect 的调用产生 WM_PAINT

消息，并在客户区显示一个新的字符串。

提供一个方法（如结构中的 bContinue 域）允许线程正常终止，这通常是一个好主意。KillThread 函数只有在正常终止线程比较困难时才应该使用，原因是线程可以分配资源，如内存等。如果当线程终止时没有释放所分配的内存，那么内存将仍然是被分配了的。线程不是进程：所分配的资源在一个进程的所有线程之间是共享的，因此当线程终止时，资源不会被自动释放。好的程序结构要求一个线程释放由它分配的所有资源。

还应该知道当第二个线程仍在运行时，可以创建第三个线程。如果 Windows 95 在 SendMessage 调用和_endthread 调用之间，将控制从第二个线程切换到第一个线程，那么窗口过程就可能响应鼠标击键而创建一个新的线程，从而出现了上述的情况。这不是什么问题，但是如果这对你自己的应用来说是一个问题，那么你可能会考虑使用临界区来避免线程之间的冲突。

14.4.2 事件对象

BIGJOB1 每当需要执行野蛮测试计算时，就创建一个线程；线程在完成计算之后自动终止。

另一种可选的方法是在程序的整个生命周期内保持线程的运行，但是只在必要时才激活它。这是一个应用事件对象的理想情况。

事件对象可以是“有信号的”（也称为“置位的”）。你可以通过下面调用来创建事件对象：

```
hEvent = CreateEvent (&sa, fManual, fInitial, pszName) ;
```

第一个参数（指向一个 SECURITY_ATTRIBUTES 结构的指针）和最后一个参数（一个事件对象的名字）只有在事件对象被多个进程共享时才有意义。在同一进程中，这些参数通常被设置为 NULL 。如果你希望事件对象被初始化为有信号的，那么将 fInitial 参数设置为 TRUE ，而如果希望事件对象被初始化为无信号的（复位的），则将 fInitial 参数设置为 FALSE 。稍候，我将讨论 fManual 参数。

为了置位一个现存的事件对象，调用：

```
SetEvent (hEvent) ;
```

为了复位一个事件对象，调用：

```
ResetEvent (hEvent) ;
```

一个程序通常调用：

```
WaitForSingleObject (hEvent, dwTimeOut) ;
```

并且将第二个参数设置为 INFINITE 。如果事件对象当前是置位的，那么函数将立即返回，否则，函数将挂起线程直到事件对象被置位。如果你将第二个参数设置为一个以毫秒计的

超时时间值，这样函数也可能在事件对象被置位之前返回。

如果最初的 CreateEvent 调用的 fManual 参数被设置为 FALSE，那么事件对象将在 WaitForSingleObject 函数返回时自动复位。这种功能特性通常使得事件对象没有必要使用 ResetEvent 函数。

现在，我们可以来看一看如图 14-5 所示的 BIGJOB2.C 程序。

BIGJOB2.MAK

```
#-----------------------
# BIGJOB2.MAK make file
#-----------------------

bigjob2.exe : bigjob2.obj
     $(LINKER) $(GUIFLAGS) -OUT:bigjob2.exe bigjob2.obj $(GUILIBS)

bigjob2.obj : bigjob2.c
     $(CC) $(CFLAGSMT) bigjob2.c
```

BIGJOB2.C

```
/*---------------------------------------
   BIGJOB2.C -- Multithreading Demo
                (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>
#include <math.h>
#include <process.h>

#define REP              100000

#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE     (WM_USER + 0)
#define WM_CALC_ABORTED  (WM_USER + 1)

Figure 14-5. The BIGJOB2 program.typedef struct
     {
     HWND   hwnd ;
     HANDLE hEvent ;
     BOOL   bContinue ;
     }
     PARAMS, *PPARAMS ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "BigJob2" ;
     HWND        hwnd ;
```

```
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Multithreading Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void Thread (PVOID pvoid)
     {
     double  A = 1.0 ;
     INT     i ;
     LONG    lTime ;
     PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;

     while (TRUE)
          {
          WaitForSingleObject (pparams->hEvent, INFINITE) ;

          lTime = GetCurrentTime () ;

          for (i = 0 ; i < REP && pparams->bContinue ; i++)
               A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;

          if (i == REP)
               {
```

```
            lTime = GetCurrentTime () - lTime ;
            SendMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
            }
        else
            SendMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;
        }
    }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
    {
    static char  *szMessage[] = { "Ready (left mouse button begins)",
                                  "Working (right mouse button aborts)",
                                  "%d repetitions in %ld msec" } ;
    static HANDLE hEvent ;
    static INT    iStatus ;
    static LONG   lTime ;
    static PARAMS params ;
    char          szBuffer[64] ;
    HDC           hdc ;
    PAINTSTRUCT   ps ;
    RECT          rect ;

    switch (iMsg)
        {
        case WM_CREATE :
            hEvent = CreateEvent (NULL, FALSE, FALSE, NULL) ;

            params.hwnd = hwnd ;
            params.hEvent = hEvent ;
            params.bContinue = FALSE ;

            _beginthread (Thread, 0, &params) ;

            return 0 ;

        case WM_LBUTTONDOWN :
            if (iStatus == STATUS_WORKING)
                {
                MessageBeep (0) ;
                return 0 ;
                }

            iStatus = STATUS_WORKING ;

            params.bContinue = TRUE ;

            SetEvent (hEvent) ;

            InvalidateRect (hwnd, NULL, TRUE) ;
            return 0 ;

        case WM_RBUTTONDOWN :
            params.bContinue = FALSE ;
            return 0 ;
```

```
          case WM_CALC_DONE :
               lTime = lParam ;
               iStatus = STATUS_DONE ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case WM_CALC_ABORTED :
               iStatus = STATUS_READY ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;

               wsprintf (szBuffer, szMessage[iStatus], REP, lTime) ;

               DrawText (hdc, szBuffer, -1, &rect,
                         DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               _endthread () ;
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 14-5 BIGJOB2程序

处理 WM_CREATE 消息时，窗口过程首先创建一个非人工(nonmanual)的被初始化为有信号的（或者说复位的）事件对象，然后它创建线程。

Thread 函数进入一个无限的 while 循环，在循环开始时首先调用 WaitForSingleObject（注意 PARAMS 结构包括一个包含事件对象句柄的域）。因为事件被初始化为复位的，所以线程被阻止在函数调用中。按下鼠标左键将导致窗口过程调用 SetEvent，这将释放由 WaitForSingleObject 调用产生的第二个线程，并开始野蛮测试计算。当计算完之后，线程再次调用 WaitForSingleObject，但是由于第一次调用已经使事件对象复位，这样，线程将被挂起，直到再次按动鼠标。

在其它方面，程序几乎和 BIGJOB1 完全一样。

14.5 线程局部存储（TLS）

多线程程序中的全局变量（以及任何被分配的内存）被程序中的所有线程共享。在一个函数中的局部静态变量也被使用函数的所有线程共享。一个函数中的局部动态变量是唯

一于各个线程的，因为它们被存储在堆栈上，而每个线程有它自己的堆栈。

有必要存在一些唯一于各个线程的持久性存储。例如，我在本章前面提到过的 C 中的 strtok 函数要求这种类型的存储。不幸的是， C 语言不支持这类存储。但是 Windows 95 中包含有四个函数，它们实现了一种技术来做到这一点，这叫做线程局部存储。

下面说明这种技术是如何工作的。首先，定义一个包含需要唯一于线程的所有数据的结构，例如：

```
typedef struct
     {
      int a;
      int b;
     }
     DATA, *PDATA;
```

主线程调用 TlsAlloc 获得一个索引值：

```
dwTlsIndex = TlsAlloc ( ) ;
```

这个值可以存储在一个全局变量中或者通过参数结构传递给线程函数。

线程函数首先为该数据结构分配内存，并使用上面所获得的索引值调用 TlsSetValue：

```
TlsSetValue (dwTlsIndex, GlobalAlloc (GPTR, sizeof (DATA)) ;
```

该函数将一个指针和某个线程及某个线程索引相关联。现在，任何需要使用这个指针的函数（包括最初的线程函数本身）都可以包含如下所示的代码：

```
PDATA pdata ;
[其它程序行]
pdata =  (PDATA)TlsGetValue (dwTlsIndex) ;
```

现在函数可以设置或者使用 pdata->a 和 pdata->b 了。在线程函数终止以前，它释放分配的内存：

```
GlobalFree (TlsGetValue (dwTlsIndex)) ;
```

当使用该数据的所有线程都终止了，主线程将释放索引：

```
TlsFree (dwTlsIndex) ;
```

这个过程刚开始可能有点令人迷惑，因此如果能看一看如何实现线程局部存储可能会有帮助（我不知道 Windows 95 实际上是如何实现的，但下面的方案是可能的）。首先，TlsAlloc 可能只是分配一块内存（长度为 0 ）并返回一个索引值，即指向这块内存的一个指针。每次使用该索引调用 TlsSetValue 时，使用 GlobalReAlloc 将内存块增大 8 个字节。在这 8 个字节中存储的是调用函数的线程 ID （通过 GetCurrentThreadId 来获得）以及传递给 TlsSetValue 函数的指针。 TlsGetValue 简单地使用线程 ID 来搜索表，然后返回指针。TlsFree 将释放内存块。

所以你看，这可能是一件容易得可以由你自己来实现的事情。不过，有工具已经为你做了，也很不错。

第 15 章　使用打印机

当为处理文本和图形而使用视频显示器时，第 3 和第 4 章中所描述的设备无关的概念看来非常完美，但对打印机和绘图仪，设备无关的概念又怎样呢？总的说来，效果也很好。在 Microsoft Windows 95 下，打印机和绘图仪有一个设备无关的图形接口。用户在为打印机编程时，不需要了解打印机控制序列和通信协议。零售的 Windows 程序显然不附带存放专门的打印机驱动程序的磁盘，通常，这些磁盘是 MS-DOS 字处理软件和图形程序必不可少的。当一个 Windows 程序包含了打印机驱动程序时，这些驱动程序一般是现存打印机驱动程序的增强版本。

在 Windows 程序中，用户可以使用同样的用于视频显示器的 GDI 函数，在打印纸上打印文本和图形，第 3 和第 4 章中所讨论的设备无关方面的许多问题（多数都与显示表面的尺寸和分辨率以及颜色容量有关）都可以用相同的方法解决。当然，一台打印机或绘图仪不像使用阴极射级管的显示器那么简单，它们使用的是打印纸。它们之间有一些比较大的差异。例如，我们从来不必考虑视频显示器没有与视频适配器连结好或者显示器出现“屏幕空间不够”的错误，但打印机脱机和缺纸则是经常会遇到的问题。

我们也不必担心视频适配器不能执行某些图形操作，更不用担心视频适配器能否处理图形，因为，如果它不能处理图形，就根本不能使用 Windows 。 但有些打印机不能打印图形（尽管它们能在 Windows 环境中使用)；绘图仪尽管可以打印矢量图形，却存在位块传输问题。

以下是其它若干需要考虑的问题：

- 打印机比视频显示器慢。尽管我们没有机会将程序性能调整到最佳状态，却不必担心视频显示器刷新所需的时间。然而，没有人想在做其它工作前一直等打印机完成打印任务。
- 程序可以用新的输出覆盖原有的显示输出，以重新使用视频显示器表面。这对打印机是不可能的，打印机只能走完一整页纸，然后在新的一页纸上打印新的内容。
- 在视频显示器上，不同的应用程序都被窗口化。而对于打印机，不同应用程序的输出必须分成不同的文档或打印作业。

要向 GDI 的其余部分中加入打印机支持功能， Windows 95 只需提供几个函数，由于找不到一个更好的名字，所以我还是称它们为打印函数。一个程序调用普通的文本和图形函数在一张纸上进行“显示”，和在屏幕上显示的方式一样。打印函数，诸如 StarDoc 、EndDoc 、 StartPAGE 和 EndPage 等负责组织输出并将它发送给打印机。

15.1 打印、后台处理和打印函数

当用户在 Windows 下使用打印机时，实际上启动了一个包含 GDI 库模块、打印驱动程序库模块（带.DRV 扩展名)、Windows 后台打印程序以及起作用的其它一些模块。在为打印机编写程序前，让我们先看一看这个过程是如何进行的。

当一个应用程序要使用打印机时，它首先使用 CreateDC 或者 PrintDlg 来获取指向打印机设备环境的一个句柄，这就使得打印机设备驱动程序库模块被装入到内存(如果还没有装入内存的话)并自己进行初始化。然后，程序调用 StartDoc 函数，通知说一个新文档开始了。StartDoc 函数是由 GDI 模块来处理的，GDI 模块调用打印机设备驱动程序中的 Control 函数告诉设备驱动程序对数据进行初始化并准备好进行打印。

打印一个文档的过程以 StartDoc 调用开始，以 EndDoc 调用结束。这两个调用对于在文档页面上书写文本或者绘制图形的 GDI 命令来说，其作用就像书挡一样。每页本身可视为一种子文件。程序调用 StartPage 来开始一页，并调用 EndPage 来结束该页。

例如，如果应用程序想在一页纸上画出一个椭圆，它首先调用 StartDoc 开始打印任务，然后再调用 StartPage 通知这是新的一页，接着调用 Ellipse，正如在屏幕上画一个椭圆一样。GDI 模块通常将 GDI 调用存储在一个磁盘文件中，通常是以增强型元文件的格式，位于由环境变量 TEMP 所指定的子目录中（如果没有 TEMP 环境变量，Windows 就将其存放在系统中第一个固定硬盘的根目录下）。该文件名以字符串~EMF(代表“增强型元文件”)开始，且以 TMP 为扩展名。然而，应该指出，打印机驱动程序可能跳过这一步。

当绘制第一页的 GDI 调用结束时，应用程序调用 EndPage 。现在，真正的工作开始了。打印机驱动程序必须把存放在元文件中的各种绘图命令翻译成打印机输出数据。绘制一页图形所需的打印机的输出数据量可能非常大，特别是当打印机没有高级页面制作语言时，更是如此。例如，一台每英寸 6 0 0 点且使用 8.5 × 11 英寸打印纸的激光打印机，如果要定义一个图形页，可能需要 4 兆以上字节的数据。

为此，打印机驱动程序经常使用一种称作“打印分带”的技术将一页分成若干称为“带”的矩形（本章后面将讨论带）。GDI 模块从打印机驱动程序获取每个带的尺度，然后设置一个与当前要处理的带相等的剪裁区，并为从元文件获得的每个绘图函数调用打印机设备驱动程序的 Output 函数，这个过程叫做“将元文件输出到设备驱动程序”。对设备驱动程序所定义的页面上的每个带，GDI 模块必须将整个元文件“输出到”设备驱动程序。这个过程完成以后，该元文件就可以删除了。

对每个带，设备驱动程序将这些绘图函数转换为在打印机上打印这些图形所需要的输出数据。这种输出数据的格式是特定于打印机的。对点阵打印机，它将是包括图形序列在内的一系列控制序列的集合（打印机驱动程序也能调用在 GDI 模块中的各种“助手”例程，用来协助这种输出的构造）。对于带有高级页面制作语言（如 PostScrpt)的激光打印机，打印机将用这种语言进行输出。

打印驱动程序将打印输出的每个带传送到 GDI 模块，随后，GDI 模块将该打印输出存入位于 TEMP 子目录的一个临时文件中，该临时文件名以字符串～ SPL 开始，并有一

个.TMP 扩展名。当处理好整个页之后，GDI 模块对后台打印程序进行一个进程间调用,通知它一个新的打印页已经准备好了。然后，应用程序就转向处理下一页。当应用程序处理完所有要打印的页后，它就调用 EndDoc 发出一个信号，表明打印作业已经完成。图 15-1 显示了应用程序、GDI 模块、打印驱动程序和后台打印程序的交互作用过程。

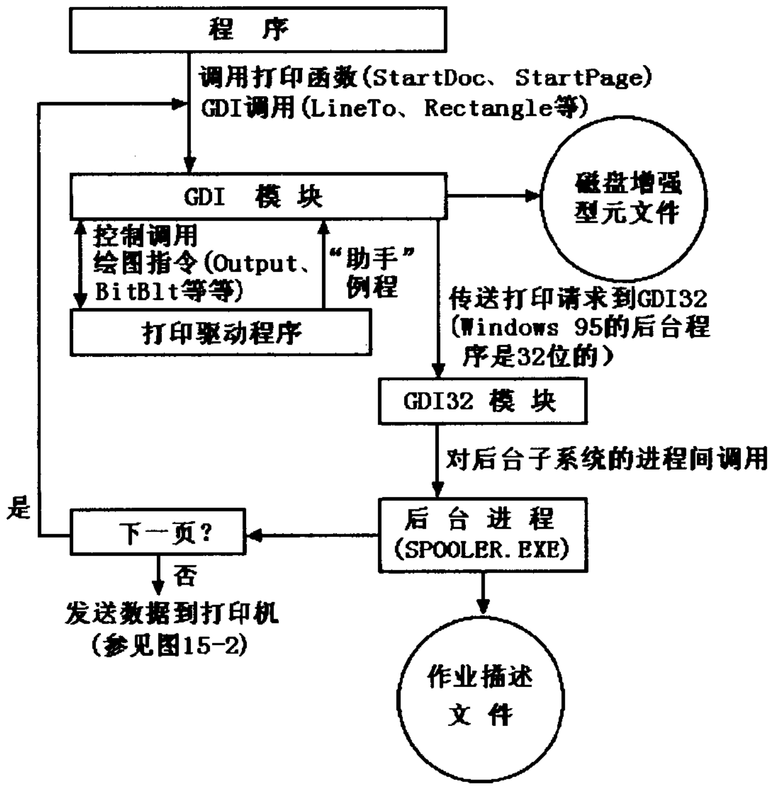


图 15-1 应用程序、GDI模块、打印驱动程序和后台程序的交互过程

Windows 95 后台打印程序实际上是几个部件的一种组合:

后台程序部件	描述
打印请求选径程序	将数据流传递给打印提供者
本地打印提供者	为本地打印机创建后台文件
网络打印提供者	为网络打印机创建后台文件
打印处理程序	将后台的设备无关数据转换为特定于目标打印机的格式
打印口监视程序	控制连接打印机的端口
语言监视程序	控制可以双向通讯的打印机，设置设备配置并检测打印机状态

后台程序可以减轻应用程序的打印负担。Windows 在启动时就装入后台打印程序，因此，当应用程序开始打印时，它已经是活动的了。当程序打印一个文件时，GDI 模块创建包含打印输出数据的文件。后台打印程序的任务是将这些文件发往打印机。GDI 模块发出一条消息来通知它一个新的打印作业开始，然后它开始读文件并将文件直接传送到打印机。为了传送这些文件，后台程序为打印机所连接的并口和串口使用各种通信函数。在后台程序向打印机发送文件的操作后，它就将包含输出数据的临时文件删除。这个过程如图 15-2 所示。

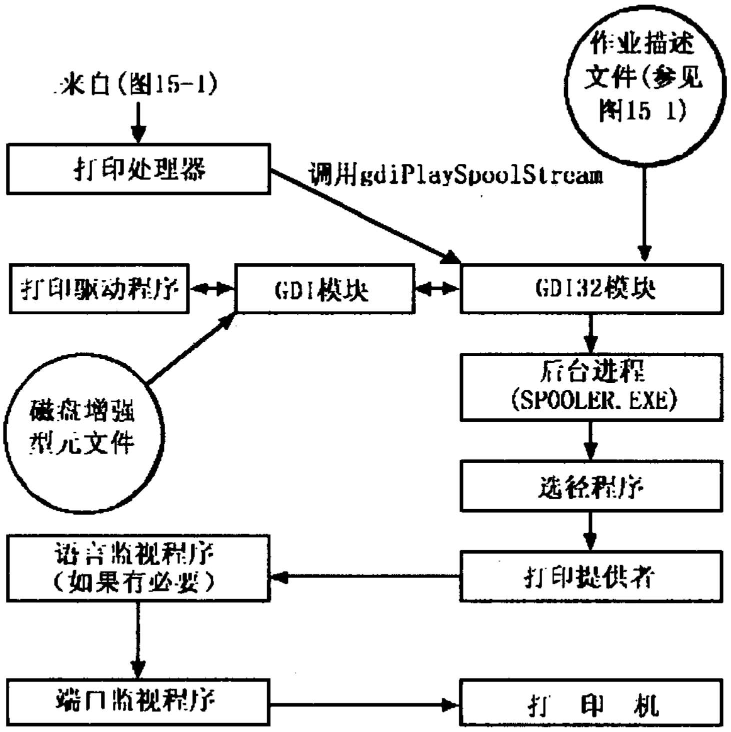


图 15-2 后台打印程序的操作过程

这个过程的大部分对应用程序是透明的。从应用程序角度来看，“打印”只发生在 GDI 模块将所有打印输出数据保存到磁盘文件中的时候，这以后(如果打印是由一个独立的线程来操作的，甚至可以在这之前)应用程序可以自由地进行其它操作。真正的文件打印操作成了后台打印程序的任务，而不是应用程序的任务。通过打印机文件夹，用户可以暂停打印作业、改变作业的优先级或取消打印作业。这种管理方式使得应用程序可能比下面这种情况“打印”得更快，即作业以实时方式打印，且必须等到打印完一页后才能处理下一页。

我们已经描述了一般的打印原理，但还有一些例外情况。其中之一是 Windows 程序要使用打印机时，并非一定需要后台打印程序。用户可以在打印机属性表格的详细资料属性页中关闭打印机的后台操作。

为什么用户希望不使用后台操作呢？因为用户可能使用了比 Windows 后台程序更快的硬件或软件后台打印程序，也可能是打印机在一个自身带有后台程序的网络上使用。一般的规律是，使用一个后台程序比使用两个后台程序更快。去掉 Windows 后台打印程序可以加快打印速度，因为打印输出数据不必保存在硬盘上，而可以直接输出到打印机，并被外部硬件或软件后台打印程序所接收。

如果没有启用 Windows 后台程序， GDI 模块就不把来自设备驱动程序的打印输出数据存入文件中，而是将这些输出数据直接输出到并行或串行打印端口。

还有另外一个例外。通常， GDI 模块将定义一页所需的所有函数存入一个增强型元文件中，然后为由驱动程序定义的每个打印分带输出一遍该元文件到打印驱动程序中。然而，如果打印驱动程序不需要打印分带的话，就不会创建这个元文件； GDI 只需简单地将绘图

函数直接送往驱动程序。进一步的变化是，应用程序也可能承担起对打印输出数据进行打印分带的责任，这就使得应用程序中的打印代码更加复杂了，但却免去了 GDI 模块创建元文件的麻烦。这样， GDI 只需简单地为每个带将函数传到打印驱动程序。

或许你现在已经发现了从一个 Windows 应用程序进行打印操作要比使用视频显示器的开销大，这样可能出现一些问题——特别是当创建元文件或打印输出文件时，如果 GDI 模块耗尽了磁盘空间。你可以更多地关心这些问题，并尝试着处理这些问题，也可以置之不理。。

下面将分析几种不同的打印方法，但让我们从头开始，先来看看如何获取打印机设备环境。

15.2 打印机设备环境

正如在视频显示器上绘图前需要得到设备环境句柄一样，在打印之前，用户必须获取一个打印机设备环境句柄。一旦有了这个句柄（并为创建一个新文档调用了 StartdDoc)，就可以用与使用视频显示设备环境句柄相同的方法来使用打印机设备环境句柄，该句柄即为前面第 3 和第 4 章中介绍的各种 GDI 调用的第一个参数。

获取打印机设备环境句柄通常是非常容易的。大多数程序简单地调用 PrintDlg 函数，除了为选择的打印机提供设备环境句柄外，这个函数还给用户提供了一个在打印之前改变打印机或者指定其他特性的机会。到现在为止，你也许只希望在缺省的打印设备上进行打印,而不需要什么对话框，就像 NOTEPAD 所做的那样。要做到这一点，你必须使用 CreateDC 函数创建打印机设备环境。

在第 4 章中，你知道如何通过如下调用来为整个视频显示器获取指向设备环境的句柄：

```
hdc = CreateDC ( “DISPLAY” NULL,NULL,NULL) ;
```

你也可以使用该函数来获取打印机设备环境句柄。然而，对打印机设备环境，第一个和第三个参数在 Win32 中被忽略了。 CreateDC 的一般语法为：

```
hdc = CreateDC (NULL, lpszDeviceName, NULL, lpIntializationData) ;
```

lpInitializationData 参数一般被置为 NULL， lpszDeviceName 参数指向一个字符串，以告诉 Windows 打印机设备的名字。在设置设备名字之前，你必须知道有哪些打印机可用。

15.2.1 获取 CreateDC 参数

一个系统可以有不只一个打印机和它相连，甚至可以有其它程序，如传真软件，将自己伪装成打印机。不论连接的打印机有多少，都只能有一个被认为是“当前的打印机”或者“缺省打印机”，这是用户从 Print 选项或打印机文件夹激活的对话框中最近一次选择的打印机。许多小的 Windows 程序只使用这个打印机来进行打印。

Windows 95 提供了几种方法来确定有哪些打印机和系统相连。我们将介绍两种方法：调用 EnumPrinters 函数和搜索 WIN.INI 文件。

15.2.1.1 EnumPrinters函数

调用 EnumPrinters 函数是找出打印机的最简单方法。函数填充一个结构数组，其中包含有关每个相连打印机的信息。你甚至可以从三种结构中进行选择，这依赖于你希望的细节程度。PRINTER_INFO_1 结构只提供打印机的名字；PRINTER_INFOR_2 提供更多的信息，包括打印驱动程序的名字、打印设备的名字和输出端口；PRINTER_INFO_5 提供设备名、端口号和超时周期。使用这个结构有更多的优点，可以使 EnumPrinters 查询登录表(registry)，而不打开任何设备，从而使操作更快。要创建打印设备环境，你只需要打印机的名字，因此 PRINTER_INFO_5 足够了。

下面是一个例子。下面的函数调用 EnumPrinters 并返回缺省打印机的设备环境句柄：

```
HDC GetPrinterDC (void)
     {
     PRINTER_INFO_5 pinfo5[3] ;
     DWORD          dwNeeded, dwReturned ;

     if (EnumPrinters (PRINTER_ENUM_DEFAULT, NULL, 5, (LPBYTE) pinfo5,
                       sizeof (pinfo5), &dwNeeded, &dwReturned))
          return CreateDC (NULL, pinfo5[0].pPrinterName, NULL, NULL) ;

     return 0 ;          // EnumPrinters failed, so return null hdc
     }
```

EnumPrinters 的第一个参数指定我们感兴趣的打印机对象的类型——在这里，我们使用 PRINTER_ENUM_DEFAULT 标志来请求只和缺省打印机有关的信息。第三个参数指示我们想填充的结构的类型—— 1 代表 PRINTER_INFO_1, 2 代表 PRINTER_INFO_2, 5 代表 PRINTER_INFO_5 。参数 pinfo5 指向一个结构数组，其中的第一个由 EnumPrinters 使用缺省打印机的设备名填充。

只会有一个缺省打印机，因此你可能奇怪为什么上面的代码要分配一个有三个 PRINTER_INFO_5 结构的数组。不幸的是，EnumPrinters 存在一个错误或者说编写得不好——看你是否宽容了。虽然它正确地填充了数组的第一个 PRINTER_INFO_5 结构，但是函数要求一个比单个结构大一些的缓冲区（不要相信文档上的说明）。将缓冲区大小增大到三倍似乎可以使 EnumPrinter 运行得更好。

如果打印机设备环境无法创建，CreateDC 函数将返回 0 。如果打印设备的名字是错误的，或者 Windows 找不到打印机驱动程序，或者输出端口名为“ none ”（表示打印机没有连到输出端口），那么都会发生这种情况。然而，用户可以使用 CreateIC 而非 CreateDC 来为一个连接到“ none ”的打印机获取一个信息环境。

假定你需要所有连接上的打印机的清单，而不只是缺省打印机，那么 EnumPrinter 也可以进行处理，虽然这要求更多一点的工作，即，只需指定 PRINTER_ENUM_LOCAL 作为第一个参数。在 Windows 95 中，这个标志导致函数枚举所有相连的打印机，既包括系统

本地的，也包括通过网络可用的。

为了枚举不知个数的打印机，你应该首先查询 EnumPrinters 以便了解它需要多大的缓冲区。它将需要的大小返回在 dwNeeded 中，然后你可以使用它来分配一个足够的缓冲区：

```
LPPRINTER_INFO_5 ppinfo5 ;
DWORD            dwNeeded, dwReturned ;
BOOL             bErr ;

// First, find out how large an array is needed
EnumPrinters (PRINTER_ENUM_LOCAL, NULL, 5, "", 0, &dwNeeded, &dwReturned) ;

// Next, allocate sufficient heap for the PRINTER_INFO_5 array
ppinfo5 = (LPPRINTER_INFO_5) HeapAlloc (GetProcessHeap (),
                                        HEAP_ZERO_MEMORY, dwNeeded) ;

// Last, call EnumPrinters again to fill the allocated PRINTER_INFO_5 array
if (ppinfo5)
     bErr = EnumPrinters (PRINTER_ENUM_LOCAL, NULL, 5, (LPBYTE) ppinfo5,
                          dwNeeded, &dwNeeded, &dwReturned)) ;
```

这个例子使用连接到系统的所有本地和远程打印机的数据来填充分配的一个 PRINTER_INFO_5 结构数组。对 EnumPrinters 的第一次调用失败，在 dwNeeded 中会返回要求的缓冲区大小。然后代码从堆上分配缓冲区，再次调用 EnumPrinters ，这次使用足够的缓冲区。EnumPrinters 填充的 PRINTER_INFO_5 结构的实际数目——一个打印机一个结构——被返回在 dwReturned 中。如果 EumPrinters 返回一个 TRUE 值，那么你可以将打印机的名字提供给用户（我们将在 DEVCAPS2 程序中这样做）。

15.2.1.2 搜索WIN.INI文件

寻找相连打印机的另一种方法要求你对 WIN.INI 文件进行搜索。在 WIN.INI 的 [windows]段列出了带有关键字 device 的缺省打印机。 CreateDC 调用所需的设备名、驱动程序名和端口名由 device 后跟的字符串指定。

```
[windows]
[其它程序行]
device=IBM Graphics,IBMGRX,LPT1
```

本例中，设备名为 IBX Graphics ，驱动程序名为 IBMGRX ，而输出端口则为 LPT1 。

下面是编写我们的 GetPrintDC 函数以便返回打印机设备环境句柄的另一种方法。这个例子使用 GetProfileString 从 WIN.INI 中获取设备串，将它分解成三个部分，然后调用 CrateDC 为缺省打印机创建一个设备环境。尽管 Win32 的 CreateDC 函数忽略驱动程序名和端口名，下面的例子还是说明了从 GetProfileString 返回的字符串中获得所有名字的方法。

```
HDC GetprinterDC ( )
     {
     char szprinter[80] ;
     char *szDevice, *szDriver, *szOutput ;
```

```
    GetProfilestring ("windows", "device",  ",,,", szprinter, 80) ;

    if (NULL != (szDevice = strtok (szprinter, ","))&&
      NULL != (szDriver = strtok (NULL,        ","))&&
      NULL != (szOutput = strtok (NULL,        ",")))
        return CreateDC (szDriver, szDevice, szOutput,NULL) ;
    return 0;
    }
```

GetProfileString 在[windows]段中查找关键字 device，并将跟在相应符号后的最多 80 个字符拷贝到 szPrinter 中(GetProfileString 的第三个参数是当 Windows 在 WIN.INI 中找不到[windows]段或关键字 device 时的缺省字符串)。字符串用普通的 C 函数 strtrok 进行分解，该函数将一个字符串分解成单词符号。注意，这里仅用了一个逗号来查找 szDevice 串的结尾，因为设备名中可以包含嵌入的空白符。由于用了一个逗号和一个空格来分隔驱动程序名和输出端口名，因此这两个名字的头尾空格均被略去（使用 strtok 来分解该字符串并不是一种完全令人满意的方法，因为 strtok 没有考虑到其它国家的 Windows 版本中可能使用多字节字符编码。如果考虑到这个因素，用户可以使用上面所述的 EnumPrinters，或者可以编写你自己的使用 CharNext 的 strtok 版本来加快对串的处理）。

WIN.INI 中的[windows]段仅列出了缺省的打印机，但更多的打印可以在[devices]段中列出。[devices]段如下所示:

```
[devices]
IBX Graphics=IBMGRX,LPT1:
Generic / Text Only=TTY,output.prn
HP Plotter=HPPLOT,COM1:
Postscript Printer=PSCRIPT,COM2:
```

每个等号左边是设备名，右边的前一个是驱动程序名，其后是输出端口。获取 WIN.INI[windows]段中指定的打印机的设备环境句柄与获取[devices]段中列出的打印机的设备环境句柄的方法大体上是相同，只是后者更困难些，因为用户有多个选择。

为了列出 WIN.INI 的[devices]段中所有的相连打印机，包括[windows]段中列出的缺省打印机，那么你应该使用 NULL 作为第二个参数来调用 GetProfileString：

```
static char szAllDevices[4096] ;
[其它程序行]
GetProfileString ("devices", NULL, "", szAllDevices,
                       sizeof szAllDevices) ;
```

返回时，szAllDevices 包含[devices]段中的一个关键字(设备名)表，除最后一个关键字以两个 NULL 结尾外，其余关键字均以一个 NULL 结尾。对前面列出的例子，szAlldevices 将包含（采用 C 的写法）：

```
IBM Graphics\0Generic / Text Only\0HP Plotter\0Postscript Printer\0\0
```

假设用户选择了其中一种设备，并且你将 szDevice 指针指向了 szAllDevices 中的相应

串，你就可以用 szDevice 来调用 CreateDC 或 CreateIC 了。如果还需要驱动器名和输出端口，那么你必须再次调用 GetProfileString 来获取该字符串的其余部分：

```
GetProfile String ("devices", szDevice, "", szPinter, 64) ;
```

必须扫描 szPrinterString 以便提取驱动程序名和输出端口名：

```
szDriver = strtok (szPrinter,",") ;
szOutput = strtok (NULL,",") ;
```

一个特定系统上的有效输出端口都被列在 WIN.INI 的[ports]段中。你不必为了使用打印机而访问 WIN.INI 中的这一段。你可以假设用户已经为打印机指定了某个端口，而且还可以进一步假设用户已经为串口（COM）正确地定义了通信参数。

[ports]段一般如下所示：

```
[ports]
LPT1:=
LPT2:=
LPT3:=
COM1:=9600,N,8,1
COM2:=1200,N,8,1
output.prn=
```

可以在这里列出 OUTPUT.PRN 文件（或任何带.PRN 扩展名的文件）以便将打印机输出定向到一个文件中。该文件名可以以一台打印机的输出端口的形式出现在 WIN.INI 的[windows]段或[devices]段中。

15.2.2 修改后的 DEVCAPS 程序

第 4 章的 DEVCAPS1 程序只显示了可以从 GetdeviceCaps 函数获得的所有信息中关于视频显示的一部分信息。图 15-3 所示的新版本的 DEVCAPS2 程序显示了关于视频显示和连接到系统的所有打印机的更多信息。它还显示一个对话框，让用户从中选择其中的打印机之一。

DEVCAPS2.MAK

```
#------------------------
# DEVCAPS2.MAK make file
#------------------------

devcaps2.exe : devcaps2.obj devcaps2.res
     $(LINKER) $(GUIFLAGS) -OUT:devcaps2.exe devcaps2.obj \
          devcaps2.res $(GUILIBS) winspool.lib

devcaps2.obj : devcaps2.c devcaps2.h
     $(CC) $(CFLAGS) devcaps2.c
```

```
devcaps2.res : devcaps2.rc devcaps2.h
     $(RC) $(RCVARS) devcaps2.rc
```

DEVCAPS2.C

```
/*---------------------------------------------------------------------
   DEVCAPS2.C -- Displays Device Capability Information (Version 2)
                 (c) Charles Petzold, 1996
  ---------------------------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "devcaps2.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
void DoBasicInfo     (HDC, HDC, int, int) ;
void DoOtherInfo     (HDC, HDC, int, int) ;
void DoBitCodedCaps  (HDC, HDC, int, int, int) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "DevCaps2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, NULL,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

```
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     static char   szDevice[32], szWindowText[64] ;
     static int    n, cxChar, cyChar,
                   nCurrentDevice = IDM_SCREEN,
                   nCurrentInfo   = IDM_BASIC ;
     static DWORD  dwNeeded, dwReturned ;
     static LPPRINTER_INFO_5 pinfo5 ;

     DWORD         i ;
     HDC           hdc, hdcInfo ;
     HMENU         hMenu ;
     PAINTSTRUCT   ps ;
     TEXTMETRIC    tm ;
     HANDLE        hPrint ;

     switch (msg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
               ReleaseDC (hwnd, hdc) ;

               lParam = 0 ;
                                                  // fall through
          case WM_WININICHANGE :
               if (lParam != 0 && lstrcmp ((PSTR) lParam, "devices") != 0)
                    return 0 ;

               hMenu = GetSubMenu (GetMenu (hwnd), 0) ;

               while (GetMenuItemCount (hMenu) > 1)
                    DeleteMenu (hMenu, 1, MF_BYPOSITION) ;

               // Get a list of all local and remote printers
               //
               // First, find out how large an array is needed; this
               //   call will fail, leaving the required size in dwNeeded
               EnumPrinters (PRINTER_ENUM_LOCAL,
                             NULL, 5, (LPBYTE) "", 0, &dwNeeded,
                             &dwReturned) ;

               // Next, allocate space for PRINTER_INFO_5 array
               if (pinfo5)
                    HeapFree (GetProcessHeap (), 0, pinfo5) ;
               pinfo5 = (LPPRINTER_INFO_5) HeapAlloc (GetProcessHeap (),
                                           HEAP_NO_SERIALIZE, dwNeeded) ;

               // Last, fill allocated PRINTER_INFO_5 array
               if (!pinfo5 || !EnumPrinters (PRINTER_ENUM_LOCAL,
```

```
                              NULL, 5, (LPBYTE) pinfo5, dwNeeded,
                              &dwNeeded, &dwReturned))
                    {
                    MessageBox (hwnd, "Could not enumerate printers!",
                                          NULL, MB_ICONSTOP) ;
                    DestroyWindow (hwnd) ;
                    return 0 ;
                    }

               for (i = 0, n = IDM_SCREEN + 1 ; i < dwReturned ; i++, n++)
                    {
                    AppendMenu (hMenu, n % 16 ? 0 : MF_MENUBARBREAK, n,
                                          pinfo5->pPrinterName) ;
                    pinfo5++ ;
                    }

               AppendMenu (hMenu, MF_SEPARATOR, 0, NULL) ;
               AppendMenu (hMenu, 0, IDM_DEVMODE, "Properties") ;

               wParam = IDM_SCREEN ;
                                          // fall through
          case WM_COMMAND :
               hMenu = GetMenu (hwnd) ;

               if (wParam < IDM_DEVMODE)          // IDM_SCREEN & Printers
                    {
                    CheckMenuItem (hMenu, nCurrentDevice, MF_UNCHECKED) ;
                    nCurrentDevice = wParam ;
                    CheckMenuItem (hMenu, nCurrentDevice, MF_CHECKED) ;
                    }
               else if (wParam == IDM_DEVMODE)    // "Properties" selection
                    {
                    GetMenuString (hMenu, nCurrentDevice, szDevice,
                                   sizeof (szDevice), MF_BYCOMMAND) ;

                    if (OpenPrinter (szDevice, &hPrint, NULL))
                         {
                         PrinterProperties (hwnd, hPrint) ;
                         ClosePrinter (hPrint) ;
                         }
                    }
               else                               // info menu items
                    {
                    CheckMenuItem (hMenu, nCurrentInfo, MF_UNCHECKED) ;
                    nCurrentInfo = wParam ;
                    CheckMenuItem (hMenu, nCurrentInfo, MF_CHECKED) ;
                    }
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;

          case WM_INITMENUPOPUP :
               if (lParam == 0)
                    EnableMenuItem (GetMenu (hwnd), IDM_DEVMODE,
                         nCurrentDevice == IDM_SCREEN ?
                              MF_GRAYED : MF_ENABLED) ;
               return 0 ;
```

```
case WM_PAINT :
     strcpy (szWindowText, "Device Capabilities: ") ;

     if (nCurrentDevice == IDM_SCREEN)
          {
          strcpy (szDevice, "DISPLAY") ;
          hdcInfo = CreateIC (szDevice, NULL, NULL, NULL) ;
          }
     else
          {
          hMenu = GetMenu (hwnd) ;
          GetMenuString (hMenu, nCurrentDevice, szDevice,
                         sizeof (szDevice), MF_BYCOMMAND) ;
          hdcInfo = CreateIC (NULL, szDevice, NULL, NULL) ;
          }

     strcat (szWindowText, szDevice) ;
     SetWindowText (hwnd, szWindowText) ;

     hdc = BeginPaint (hwnd, &ps) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;

     if (hdcInfo)
          {
          switch (nCurrentInfo)
               {
               case IDM_BASIC :
                    DoBasicInfo (hdc, hdcInfo, cxChar, cyChar) ;
                    break ;

               case IDM_OTHER :
                    DoOtherInfo (hdc, hdcInfo, cxChar, cyChar) ;
                    break ;

               case IDM_CURVE :
               case IDM_LINE :
               case IDM_POLY :
               case IDM_TEXT :
                    DoBitCodedCaps (hdc, hdcInfo, cxChar, cyChar,
                         nCurrentInfo - IDM_CURVE) ;
                    break ;
               }
          DeleteDC (hdcInfo) ;
          }

     EndPaint (hwnd, &ps) ;
     return 0 ;

case WM_DESTROY :
     if (pinfo5)
          HeapFree (GetProcessHeap (), 0, pinfo5) ;

     PostQuitMessage (0) ;
     return 0 ;
}
```

```
     return DefWindowProc (hwnd, msg, wParam, lParam) ;
     }

void DoBasicInfo (HDC hdc, HDC hdcInfo, int cxChar, int cyChar)
     {
     static struct
          {
          int   nIndex ;
          char *szDesc ;
          }
          info[] =
          {
          HORZSIZE,        "HORZSIZE     Width in millimeters:",
          VERTSIZE,        "VERTSIZE     Height in millimeters:",
          HORZRES,         "HORZRES      Width in pixels:",
          VERTRES,         "VERTRES      Height in raster lines:",
          BITSPIXEL,       "BITSPIXEL    Color bits per pixel:",
          PLANES,          "PLANES       Number of color planes:",
          NUMBRUSHES,      "NUMBRUSHES   Number of device brushes:",
          NUMPENS,         "NUMPENS      Number of device pens:",
          NUMMARKERS,      "NUMMARKERS   Number of device markers:",
          NUMFONTS,        "NUMFONTS     Number of device fonts:",
          NUMCOLORS,       "NUMCOLORS    Number of device colors:",
          PDEVICESIZE,     "PDEVICESIZE  Size of device structure:",
          ASPECTX,         "ASPECTX      Relative width of pixel:",
          ASPECTY,         "ASPECTY      Relative height of pixel:",
          ASPECTXY,        "ASPECTXY     Relative diagonal of pixel:",
          LOGPIXELSX,      "LOGPIXELSX   Horizontal dots per inch:",
          LOGPIXELSY,      "LOGPIXELSY   Vertical dots per inch:",
          SIZEPALETTE,     "SIZEPALETTE  Number of palette entries:",
          NUMRESERVED,     "NUMRESERVED  Reserved palette entries:",
          COLORRES,        "COLORRES     Actual color resolution:"
          } ;
     char   szBuffer[80] ;
     int    i ;

     for (i = 0 ; i < sizeof (info) / sizeof (info[0]) ; i++)
          TextOut (hdc, cxChar, (i + 1) * cyChar, szBuffer,
               sprintf (szBuffer, "%-40s%8d", info[i].szDesc,
                    GetDeviceCaps (hdcInfo, info[i].nIndex))) ;
     }

void DoOtherInfo (HDC hdc, HDC hdcInfo, int cxChar, int cyChar)
     {
     static BITS clip[] =
          {
          CP_RECTANGLE,  "CP_RECTANGLE",      "Can Clip To Rectangle:"
          } ;

     static BITS raster[] =
          {
          RC_BITBLT,         "RC_BITBLT",         "Capable of simple BitBlt:",
          RC_BANDING,        "RC_BANDING",        "Requires banding support:",
          RC_SCALING,        "RC_SCALING",        "Requires scaling support:",
          RC_BITMAP64,       "RC_BITMAP64",       "Supports bitmaps >64K:",
          RC_GDI20_OUTPUT,   "RC_GDI20_OUTPUT",   "Has 2.0 output calls:",
```

```
          RC_DI_BITMAP,     "RC_DI_BITMAP",     "Supports DIB to memory: ",
          RC_PALETTE,       "RC_PALETTE",       "Supports a palette: ",
          RC_DIBTODEV,      "RC_DIBTODEV",      "Supports bitmap conversion: ",
          RC_BIGFONT,       "RC_BIGFONT",       "Supports fonts >64K: ",
          RC_STRETCHBLT,    "RC_STRETCHBLT",    "Supports StretchBlt: ",
          RC_FLOODFILL,     "RC_FLOODFILL",     "Supports FloodFill: ",
          RC_STRETCHDIB,    "RC_STRETCHDIB",    "Supports StretchDIBits: "
          } ;

     static char *szTech[] = { "DT_PLOTTER (Vector plotter)",
                               "DT_RASDISPLAY (Raster display)",
                               "DT_RASPRINTER (Raster printer)",
                               "DT_RASCAMERA (Raster camera)",
                               "DT_CHARSTREAM (Character-stream, PLP)",
                               "DT_METAFILE (Metafile, VDM)",
                               "DT_DISPFILE (Display-file)" } ;
     char        szBuffer[80] ;
     int         i ;

     TextOut (hdc, cxChar, cyChar, szBuffer,
          sprintf (szBuffer, "%-24s%04XH",
               "DRIVERVERSION:", GetDeviceCaps (hdcInfo, DRIVERVERSION))) ;

     TextOut (hdc, cxChar, 2 * cyChar, szBuffer,
          sprintf (szBuffer, "%-24s%-40s",
               "TECHNOLOGY:",szTech[GetDeviceCaps (hdcInfo, TECHNOLOGY)])) ;

     TextOut (hdc, cxChar, 4 * cyChar, szBuffer,
          sprintf (szBuffer, "CLIPCAPS (Clipping capabilities)")) ;

     for (i = 0 ; i < sizeof (clip) / sizeof (clip[0]) ; i++)
          TextOut (hdc, 9 * cxChar, (i + 6) * cyChar, szBuffer,
               sprintf (szBuffer, "%-16s%-28s %3s",
                    clip[i].szMask, clip[i].szDesc,
                    GetDeviceCaps (hdcInfo, CLIPCAPS) & clip[i].nMask ?
                         "Yes" : "No")) ;

     TextOut (hdc, cxChar, 8 * cyChar, szBuffer,
          sprintf (szBuffer, "RASTERCAPS (Raster capabilities)")) ;

     for (i = 0 ; i < sizeof (raster) / sizeof (raster[0]) ; i++)
          TextOut (hdc, 9 * cxChar, (i + 10) * cyChar, szBuffer,
               sprintf (szBuffer, "%-16s%-28s %3s",
                    raster[i].szMask, raster[i].szDesc,
                    GetDeviceCaps (hdcInfo, RASTERCAPS) & raster[i].nMask ?
                         "Yes" : "No")) ;
     }

void DoBitCodedCaps (HDC hdc, HDC hdcInfo, int cxChar,
                                           int cyChar, int nType)
     {
     static BITS curves[] =
          {
          CC_CIRCLES,     "CC_CIRCLES",     "circles: ",
          CC_PIE,         "CC_PIE",         "pie wedges: ",
          CC_CHORD,       "CC_CHORD",       "chord arcs: ",
```

```
        CC_ELLIPSES,     "CC_ELLIPSES",     "ellipses: ",
        CC_WIDE,         "CC_WIDE",         "wide borders: ",
        CC_STYLED,       "CC_STYLED",       "styled borders: ",
        CC_WIDESTYLED,   "CC_WIDESTYLED",   "wide and styled borders: ",
        CC_INTERIORS,    "CC_INTERIORS",    "interiors: "
        } ;

static BITS lines[] =
        {
        LC_POLYLINE,     "LC_POLYLINE",     "polyline: ",
        LC_MARKER,       "LC_MARKER",       "markers: ",
        LC_POLYMARKER,   "LC_POLYMARKER",   "polymarkers",
        LC_WIDE,         "LC_WIDE",         "wide lines: ",
        LC_STYLED,       "LC_STYLED",       "styled lines: ",
        LC_WIDESTYLED,   "LC_WIDESTYLED",   "wide and styled lines: ",
        LC_INTERIORS,    "LC_INTERIORS",    "interiors: "
        } ;

static BITS poly[] =
        {
        PC_POLYGON,      "PC_POLYGON",      "alternate fill polygon: ",
        PC_RECTANGLE,    "PC_RECTANGLE",    "rectangle: ",
        PC_WINDPOLYGON,"PC_WINDPOLYGON","winding number fill polygon: ",
        PC_SCANLINE,     "PC_SCANLINE",     "scanlines: ",
        PC_WIDE,         "PC_WIDE",         "wide borders: ",
        PC_STYLED,       "PC_STYLED",       "styled borders: ",
        PC_WIDESTYLED,   "PC_WIDESTYLED",   "wide and styled borders: ",
        PC_INTERIORS,    "PC_INTERIORS",    "interiors: "
        } ;

static BITS text[] =
        {
        TC_OP_CHARACTER, "TC_OP_CHARACTER", "character output precision: ",
        TC_OP_STROKE,    "TC_OP_STROKE",    "stroke output precision: ",
        TC_CP_STROKE,    "TC_CP_STROKE",    "stroke clip precision: ",
        TC_CR_90,        "TC_CP_90",
                               "90 degree character rotation: ",
        TC_CR_ANY,       "TC_CR_ANY",       "any character rotation: ",
        TC_SF_X_YINDEP,  "TC_SF_X_YINDEP",
                               "scaling independent of X and Y: ",
        TC_SA_DOUBLE,    "TC_SA_DOUBLE",
                               "doubled character for scaling: ",
        TC_SA_INTEGER,   "TC_SA_INTEGER",
                               "integer multiples for scaling: ",
        TC_SA_CONTIN,    "TC_SA_CONTIN",
                               "any multiples for exact scaling: ",
        TC_EA_DOUBLE,    "TC_EA_DOUBLE",  "double weight characters: ",
        TC_IA_ABLE,      "TC_IA_ABLE",    "italicizing: ",
        TC_UA_ABLE,      "TC_UA_ABLE",    "underlining: ",
        TC_SO_ABLE,      "TC_SO_ABLE",    "strikeouts: ",
        TC_RA_ABLE,      "TC_RA_ABLE",    "raster fonts: ",
        TC_VA_ABLE,      "TC_VA_ABLE",    "vector fonts: "
        } ;

static struct
        {
```

```
          int   nIndex ;
          char  *szTitle ;
          BITS  (*pbits) [] ;
          short nSize ;
          }
          bitinfo[] =
          {
          CURVECAPS,   "CURVCAPS (Curve Capabilities)",
                       (BITS (*) []) curves, sizeof (curves) /
                                             sizeof (curves[0]),
          LINECAPS,    "LINECAPS (Line Capabilities)",
                       (BITS (*) []) lines, sizeof (lines) /
                                            sizeof (lines[0]),
          POLYGONALCAPS, "POLYGONALCAPS (Polygonal Capabilities)",
                       (BITS (*) []) poly, sizeof (poly) / sizeof (poly[0]),
          TEXTCAPS,    "TEXTCAPS (Text Capabilities)",
                       (BITS (*) []) text, sizeof (text) / sizeof (text[0])
          } ;

     static char szBuffer[80] ;
     BITS (*pbits) [] = bitinfo[nType].pbits ;
     int  nDevCaps = GetDeviceCaps (hdcInfo, bitinfo[nType].nIndex) ;
     int  i ;

     TextOut (hdc, cxChar, cyChar, bitinfo[nType].szTitle,
              strlen (bitinfo[nType].szTitle)) ;

     for (i = 0 ; i < bitinfo[nType].nSize ; i++)
          TextOut (hdc, cxChar, (i + 3) * cyChar, szBuffer,
               sprintf (szBuffer, "%-16s %s %-32s %3s",
                    (*pbits)[i].szMask, "Can do", (*pbits)[i].szDesc,
                    nDevCaps & (*pbits)[i].nMask ? "Yes" : "No")) ;
     }
```

DEVCAPS2.H

```
/*------------------------
   DEVCAPS2.H header file
  ------------------------*/

#define IDM_SCREEN  1

#define IDM_DEVMODE 0x100

#define IDM_BASIC   0x101
#define IDM_OTHER   0x102
#define IDM_CURVE   0x103
#define IDM_LINE    0x104
#define IDM_POLY    0x105
#define IDM_TEXT    0x106

typedef struct
     {
     short  nMask ;
     char   *szMask ;
     char   *szDesc ;
     }
```

```
    BITS ;
```

DEVCAPS2.RC

```
/*----------------------------
   DEVCAPS2.RC resource script
  ----------------------------*/

#include "devcaps2.h"

DevCaps2 MENU
     {
     POPUP "&Device"
          {
          MENUITEM "&Screen",                   IDM_SCREEN, CHECKED
          }
     POPUP "&Capabilities"
          {
          MENUITEM "&Basic Information",        IDM_BASIC, CHECKED
          MENUITEM "&Other Information",        IDM_OTHER
          MENUITEM "&Curve Capabilities",       IDM_CURVE
          MENUITEM "&Line Capabilities",        IDM_LINE
          MENUITEM "&Polygonal Capabilities",   IDM_POLY
          MENUITEM "&Text Capabilities",        IDM_TEXT
          }
     }
```

图 15-3 DEVCAPS程序

因为 DEVCAPS2 只获取打印机的信息环境，所以即使其输出端口标有“ none ”标记，用户仍然可以从 DEVCAPS2 的菜单中选择所需打印机。如果用户想比较不同打印机的功能，可以先用打印机文件夹增加各种打印驱动程序。

15.2.3 PrintProperties 调用

DEVCAP2 的 Device 菜单上还有一个称为 Properties 的选项。要使用这个选项，首先从 Device 菜单中选择一个打印机，然后再选择 Properties ，这时弹出一个对话框。对话框从何而来呢？它由打印机驱动程序调用，而且至少还需要用户选择纸的尺寸。大多数打印机驱动也可以让用户在“肖像(portrait)”或“风景(landsacpe)”模式中进行选择。在肖像模式（一般为缺省模式）下纸的短边是顶部；在风景模式下，纸的长边是顶部。如果改变该模式，则所作的改变将在 DEVCAPS2 程序从 GetDeviceCaps 函数获取的信息中反映出来：水平尺寸和分辨率将与垂直尺寸和分辨率交换。彩色绘图仪的 Properties 对话框内容十分广泛，它们要求用户输入安装在绘图仪上的笔的颜色和使用的绘图纸(或透明胶片)的型号。

Windows 95 中的所有打印机驱动程序都包含一个称为 ExDeviceMode 的输出函数，它调用对话框并保存用户输入的信息。有些打印机驱动程序也将这些信息保存在登录表中或者保存在 WIN.INI 文件中自己的段里，有些则不然。那些保存信息的打印机驱动程序在下次运行 Windows 时将访问该信息。

允许用户选择打印机的 Windows 程序通常只调用 PrintDlg ，这个有用的函数在准备打

印时负责和用户之间的所有通讯工作，并负责处理用户要求的所有改变。当用户按下 Properties 按钮时，PrintDlg 还会激活属性表格对话框。在 WORDPAD 中试试，你将发现它和 DEVCAPS2 显示的那个对话框一样。

程序还可以通过直接调用打印机驱动程序的 ExtDeviceMode 或者 ExtDeviceModePropSheet 函数来显示打印机的属性对话框，然而，我不鼓励你这样做。通过调用 PrinterProperties 来激活对话框要好得多，这个 GDI 函数可以访问 16 位的打印机驱动程序，而用户的 32 位程序却不能（除非用户想编写自己的 thunking 层）。

PrinterProperties 要求打印机对象的句柄，你可以通过 OpenPrinter 函数来得到。当用户取消属性表格对话框时，PrinterProperties 返回，然后用户通过调用 ClosePrinter，释放打印机句柄。DEVCAPS 就是这样做到这一点的。

程序首先取得刚刚在 Device 菜单中选择的打印机的名字，并将其存入一个名为 szDevice 的字符数组中。

```
GetMenuString (hMenu, nCurrentDevice, szDevice,
               sizeof (szDevice), MF_BYCOMMAND) ;
```

然后，使用 OpenPrinter 获得该设备的句柄。如果调用成功，那么程序接着调用 PrinterProperties 激活对话框，然后调用 ClosePrinter 释放设备句柄：

```
if (OpenPrinter (szDevice, &hPrint, NULL))
     {
     PrinterProperties (hwnd, hPrint) ;
     ClosePrinter (hPrint) ;
     }
```

15.2.4 检查 BitBlt 支持

你可以用 GetDeviceCaps 函数来获取页中可打印区的尺寸和分辨率（通常，该区域不能与整张纸的大小相同）。如果用户想自己进行缩放操作，也可以获得相对的象素宽度和高度。

以 RASTERCAPS（“光栅支持”）参数调用 GetDeviceCaps，它返回的 RC_BITBLT 位包含了另一个重要的打印机特性，该位指示设备是否能进行位块传送。大多数点阵和激光打印机都能进行位块传送，而大多数绘图仪却不能。不能处理位块传送的设备不支持下列 GDI 函数：CreateCompatibleDC、CreateCompatibleBitmap、PatBlt、StretchBlt、GrayString、Drawicon、SetPixel、GetPixel、FloodFill、ExtFlootFill、FillRgn、InvertRgn、PaintRgn、FillRect、FillRect、FrameRect 和 InvertRect。这是在视频显示器上使用 GDI 函数与在打印机上使用它们的唯一重要区别。

15.3 FORMFEED 程序

现在可以开始打印了，我们尽可能简单地开始。事实上，我们的第一个程序只是让打

印机走纸而已。图 15-4 所示的 FORMFEED 程序打印所需的最小需求。

FORMFEED.MAK

```
#-----------------------
# FORMFEED.MAK make file
#-----------------------

formfeed.exe: formfeed.obj
     $(LINKER) $(GUIFLAGS) -OUT:formfeed.exe formfeed.obj \
     $(GUILIBS) winspool.lib

formfeed.obj: formfeed.c
     $(CC) $(CFLAGS) formfeed.c
```

FORMFEED.C

```
/*---------------------------------------------
   FORMFEED.C -- Advances printer to next page
                 (c) Charles Petzold, 1996
  ---------------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC (void) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine, int iCmdShow)
     {
     static DOCINFO di      = { sizeof (DOCINFO), "FormFeed", NULL } ;
     HDC            hdcPrint = GetPrinterDC () ;

     if (hdcPrint != NULL)
          {
          if (StartDoc (hdcPrint, &di) > 0)
               if (StartPage (hdcPrint) > 0 && EndPage (hdcPrint) > 0)
                    EndDoc (hdcPrint) ;

          DeleteDC (hdcPrint) ;
          }
     return FALSE ;
     }

HDC GetPrinterDC (void)
     {
     PRINTER_INFO_5 pinfo5[3] ;
     DWORD          dwNeeded, dwReturned ;

     if (EnumPrinters (PRINTER_ENUM_DEFAULT, NULL, 5, (LPBYTE) pinfo5,
                       sizeof (pinfo5), &dwNeeded, &dwReturned))
          return CreateDC (NULL, pinfo5[0].pPrinterName, NULL, NULL) ;

     return 0 ;            // EnumPrinters failed, so return null hdc
     }
```

图 15-4　FORMFEED程序

FORMFEED 包含以前出现过的 GetPrinterDC 函数。除了获取打印机设备环境（然后再删除它）外，程序只调用了我们在本章前面讨论过的四个打印函数。FORMFEED 首先调用 StartDoc 开始一个新的文件，它测试从 StartDoc 返回的值，只有返回值是正数时，才继续下去:

```
if (StartDoc (hdcPrint, &di) > 0)
```

DOCINFO 结构的第二个域 di 指向一个串，该串标识正在被打印的文件。当文件正在被打印或者在等待打印时，这个串将出现在打印机任务队列中的 Document Name 列中（为了看到排队的打印文件的清单，按下开始按钮，选择设置，然后选择打印机，并双击打印机图标）。通常，该串包含进行打印的应用程序名和被打印的文件名。本例中，它就是“ FormFeed ”。

如果 StartDoc 成功(由一个正的返回值表示)，那么 FORMFEED 调用 StartPage ，紧接着立即调用 EndPage 。这一过程将打印机推进到新的一页，再次对返回值进行测试:

```
if (StartPage (hdcPrint) > 0 && EndPage (hdcPrint) > 0)
```

最后，如果不出错，文档就结束:

```
EndDoc (hdcPrint) ;
```

要注意的是，只有当不出错时，才调用 EndDoc 函数。如果其它打印函数中的某一个返回错误代码，那么 GDI 实际上已经中断了文档的打印。如果打印机当前未打印，这种错误代码通常会使打印机复位。

测试打印函数的返回值是检测错误的最简单方法。如果你想向用户报告错误，就必须调用 GetLastError 来确定错误。（在本章后面，我们将看到使用 GetLastError 来报告打印机错误。）例如，如果 GDI 无法找到足够的磁盘空间来存储触发打印机走纸的必要打印输出，就会有错误发生。对大多数打印机来说，这是最不可能的事。不过，想找乐子的话，你可以指定 PostScript 打印机驱动程序作为当前打印机，输出端口设为 OUTPUT.PRN 。运行 FORMFEED ，然后检查文件大小(它将超过 8KB)。

如果你写过 MS-DOS 下的简单走纸程序，就应该知道，对于大多数打印机， ASCII 码 12 启动走纸。为什么不简单地使用 GDI 函数 OpenPrinter 打开打印口，然后用 Write 输出 ASCII 码 12 呢?当然，你完全可以这么做，但是必须确定打印机连接的是串行口还是并行口。通过使用 OpenPrinter 返回的打印机标识符来调用 GetPrinter ，并从 PRINTER_INFO_5 结构中读出端口标识符可以做到这一点。然后你还要确定另外的程序是不是正在使用打印机。你并不希望在文档的中间出现走纸，对不对？最后，你还必须确定 ASCII 码 12 是不是所连接打印机的走纸符，因为并非所有打印机的走纸符都是 12 。事实上，在 PostScript 中的走纸命令就不是 12 ，而是单词 showpage 。

简单地说，不要试图绕过 Windows ；而应该坚持在打印中使用 Windows 函数。

15.4 打印图形和文本

在一个 Windows 程序中，打印所需的额外开销通常比 FORMFEED 程序高得多，而且

还要用 GDI 函数来实际打印一些东西。我们来写一个打印一页文本和图形的程序，采用 FORMFEED 程序中的方法，并加入一些新东西。该程序将有四个版本 PRINT1 、 PRINT2 、 PRINT3 和 PRINT4 ，为避免代码重复，每个程序都用 PRINT.C 文件中的函数，如图 15-5 所示。

PRINT.C

```
/*-------------------------------------------------------------------
   PRINT.C -- Common routines for Print1, Print2, Print3, and Print4
  -------------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
BOOL PrintMyPage (HWND) ;

extern HINSTANCE hInst ;
extern char      szAppName[] ;
extern char      szCaption[] ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND       hwnd ;
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hInst = hInstance ;

     hwnd = CreateWindow (szAppName, szCaption,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
```

```
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

HDC GetPrinterDC (void)
     {
     PRINTER_INFO_5 pinfo5[3] ;
     DWORD          dwNeeded, dwReturned ;

     if (EnumPrinters (PRINTER_ENUM_DEFAULT, NULL, 5, (LPBYTE) pinfo5,
                       sizeof (pinfo5), &dwNeeded, &dwReturned))
          return CreateDC (NULL, pinfo5[0].pPrinterName, NULL, NULL) ;

     return 0 ;              // EnumPrinters failed, so return null hdc
     }

void PageGDICalls (HDC hdcPrn, int cxPage, int cyPage)
     {
     static char szTextStr[] = "Hello, Printer!" ;

     Rectangle (hdcPrn, 0, 0, cxPage, cyPage) ;

     MoveToEx (hdcPrn, 0, 0, NULL) ;
     LineTo   (hdcPrn, cxPage, cyPage) ;
     MoveToEx (hdcPrn, cxPage, 0, NULL) ;
     LineTo   (hdcPrn, 0, cyPage) ;

     SaveDC (hdcPrn) ;

     SetMapMode       (hdcPrn, MM_ISOTROPIC) ;
     SetWindowExtEx   (hdcPrn, 1000, 1000, NULL) ;
     SetViewportExtEx (hdcPrn, cxPage / 2, -cyPage / 2, NULL) ;
     SetViewportOrgEx (hdcPrn, cxPage / 2,  cyPage / 2, NULL) ;

     Ellipse (hdcPrn, -500, 500, 500, -500) ;

     SetTextAlign (hdcPrn, TA_BASELINE | TA_CENTER) ;

     TextOut (hdcPrn, 0, 0, szTextStr, sizeof (szTextStr) - 1) ;

     RestoreDC (hdcPrn, -1) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     static int   cxClient, cyClient ;
     HDC          hdc ;
     HMENU        hMenu ;
     PAINTSTRUCT  ps ;

     switch (msg)
          {
          case WM_CREATE :
               hMenu = GetSystemMenu (hwnd, FALSE) ;
```

```
               AppendMenu (hMenu, MF_SEPARATOR, 0, NULL) ;
               AppendMenu (hMenu, 0, 1, "&Print") ;
               return 0 ;

          case WM_SIZE :
               cxClient = LOWORD (lParam) ;
               cyClient = HIWORD (lParam) ;
               return 0 ;

          case WM_SYSCOMMAND :
               if (wParam == 1)
                    {
                    if (PrintMyPage (hwnd))
                         MessageBox (hwnd, "Could not print page!",
                              szAppName, MB_OK | MB_ICONEXCLAMATION) ;
                    return 0 ;
                    }
               break ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               PageGDICalls (hdc, cxClient, cyClient) ;

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, msg, wParam, lParam) ;
```

图 15-5 PRINT1、PRINT2、PRINT3和PRINT4程序的PRINT.C文件

PRINT.C 包括函数 WinMain 、 WndProc 、 GetPrintDC 以及一个称为 PageGDICalls 的函数。 PageGDICalls 函数接收打印机设备环境句柄和两个包含打印页宽度及高度的变量。这个函数还负责画一个包围整个页的矩形，有两条对角线，页中间有一个椭圆（其直径是打印机高度和宽度中较小的那个的一半），文本“ Hello, Printer!”位于椭圆中间。

处理 WM_CREATE 消息时， WndProc 将一个 Print 选项加到系统菜单上。 选择该选项将调用 PrintMypage ，此函数的功能在程序的四个版本中将不断增强。当打印遇到错误时， PrintMyPage 返回 TRUE （非零）值，其余情况下返回 FALSE 。如果 PrintMyPage 返回 TRUE ， WndProc 就会显示一个消息框以告知用户发生了错误。

15.4.1 打印的基本过程

打印程序的第一个版本是 PRINT1 ，见图 15-6 。经编译后即可运行此程序，然后从系统菜单中选 Print 。如果 MS-DOS 环境变量 TEMP 指定了硬盘(或没有 TEMP 变量)，那么当 GDI 模块把打印输出送到临时文件中时可以看到磁盘活动。当 PRINT1 终止后，后台程

序应该开始向打印机发送磁盘文件。

让我们来看看 PRINT1.C 的代码。如果 PrintMyPage 不能获取打印机的设备环境句柄，它就返回 TRUE，且 WndProc 显示消息框来表示出错。如果函数成功地获取了设备环境句柄，它就通过调用 GetDeviceCaps 来了解打印页的水平和垂直尺寸（单位为象素）：

```
xPage=GetDeviceCaps (hdcPrn, HORZRES) ;
yPage=GetDeviceCAPS (hdcprn, VERTRES) ;
```

这并不是打印纸的尺寸，而是可打印区域的尺寸。在进行该调用后，PRINT1 中 PrintMyPage 函数代码在结构上与 FORMFEED 中的代码相同，不同点只是 PRINT1 在 StartPage 与 EndPage 调用之间调用了 PageGDICalls。仅当 StartDoc、StartPage 和 EndPage 调用成功时，PRINT1 才调用 EndDoc 打印函数。

PRINT1.MAK

```
#---------------------
# PRINT1.MAK make file
#---------------------

print1.exe : print.obj print1.obj
     $(LINKER) $(GUIFLAGS) -OUT:print1.exe print.obj print1.obj \
     $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print1.obj : print1.c
     $(CC) $(CFLAGS) print1.c
```

PRINT1.C

```
/*-------------------------------------
   PRINT1.C -- Bare-Bones Printing
               (c) Charles Petzold, 1996
  -------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC (void) ;              // in PRINT.C
void PageGDICalls (HDC, int, int) ;

HINSTANCE hInst ;
char      szAppName[] = "Print1" ;
char      szCaption[] = "Print Program 1" ;

BOOL PrintMyPage (HWND hwnd)
     {
     static DOCINFO di = { sizeof (DOCINFO), "Print1: Printing", NULL } ;
     BOOL           bError = FALSE ;
     HDC            hdcPrn ;
     int            xPage, yPage ;

     if (NULL == (hdcPrn = GetPrinterDC ()))
```

```
          return TRUE ;

     xPage = GetDeviceCaps (hdcPrn, HORZRES) ;
     yPage = GetDeviceCaps (hdcPrn, VERTRES) ;

     if (StartDoc (hdcPrn, &di) > 0)
          {
          if (StartPage (hdcPrn) > 0)
               {
               PageGDICalls (hdcPrn, xPage, yPage) ;

               if (EndPage (hdcPrn) > 0)
                    EndDoc (hdcPrn) ;
               else
                    bError = TRUE ;
               }
          }
     else
          bError = TRUE ;

     DeleteDC (hdcPrn) ;
     return bError ;
     }
```

图 15-6 PRINT1程序

如果你在包含 TEMP 子目录的驱动器上运行 PRINT1，而 TEMP 缺少足够的空间来存储图形输出的整个页，那么你将看到 Windows 的一个错误消息： Windows could not write to the printer spool file. Make sure your disk has enough free space and try again.（ Windows 无法写入后台打印文件。确保你的磁盘有足够的空闲空间，并再试一次）。

这并不是什么灾难性的问题——你只需要点一下 OK 按钮来清除这个消息。然后可以删除一些没用的文件，在磁盘中清理出一些空间来，正如消息中所建议的那样。然而，如果是当你快速地连续提交一些打印任务时发生了错误，那么只要在再次打印之前等待一会儿就可能避免这个问题。这是因为这种错误可能是由于 GDI 打印函数创建的其它临时文件占用了 TEMP 子目录而引起的。给后台程序足够的时间将这些文件发送给打印机然后清除它们，当前正在打印的程序就可以继续下去了。

15.4.2 使用放弃过程来取消打印

如果我们不小心在 PRINT1 中引入了一个错误，导致打印机吐出一叠而不是一张纸时，怎么办呢？当打印机开始毫不吝啬地吐纸时我们可以做什么呢？当然，我们总是可以拔下电源。我们也可以从打印机文件夹中取消打印，通过从开始菜单中选择设置，再选择打印机，双击打印机图标，并从打印机菜单中选择删除任务来中止打印。

正在打印时，一个程序应该给用户提供一种转移的方便。当后台被允许时，从打印程序中取消打印的机会不会持续太长——只有在后台接收到数据并创建了一个后台文件之前。在这之后，程序对打印过程不再拥有控制权了（然而，用户仍然可以从 Printer 中取消打印任务）。如果后台被禁用，那么程序可以在打印机的缓冲区接收到最后的打印数据之

前的任何时候中止打印。

在一个程序内取消一个打印任务要求一种被称为“放弃过程”的技术。放弃过程在程序中只是个较小的输出函数，用户可以使用 SetAbortProc 函数将该函数的地址传给 Windows 。然后 GDI 在打印时，重复调用该过程，不断地问：“我是否应该继续打印？”

我们看看将放弃过程加到打印逻辑中去需要些什么，然后检查一些分支。放弃过程命名为 AbortProc ，其形式为：

```
Bool CALLBACK AbortProc (HDC hdcPrn, int iCode)
     {
     [其它程序行]
     }
```

打印前，你必须通过调用 SetAbortProc 来登记放弃过程：

```
SetAbortProc (hdcPrn, AbortProc) ;
```

在调用 StartDoc 前调用上面的函数，打印完成后没有必要清除放弃过程。

在处理 EndPage 调用时(亦即，在将元文件放入设备驱动程序并创建临时打印文件时)，GDI 常常调用放弃过程。参数 hdcPrn 是打印机设备环境句柄。如果一切正常， iCode 参数是 0 ，如果 GDI 模块在生成临时文件时耗尽了磁盘空间， iCode 就是 SP_OUTOFDISK 。

如果打印作业继续，那么 AbortProc 必须返回 TRUE (非零)；如果打印作业异常结束，就返回 FALSE(零)。放弃过程可以被简化为如下所示的形式：

```
Bool CALLBACK AbortProc (HDC hdcPrn, int iCode)
     {
     MSG msg ;
     While (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          TranslateMesage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return TRUE;
     }
```

这个函数看起来有点特殊，其实它看起来像是消息循环。用户会注意到，这个“消息循环”调用 PeekMessage 而不是 GetMessage 。我在第 4 章和 RANDRECT 程序一起讨论过 PeekMessage 。还可以记起， PeekMessage 将控制返回给程序，而不管程序的消息队列中是否有消息存在。

只要 PeekMessage 返回 TRUE ，那么 AbortProc 函数中的消息循环就重复调用 PeekMessage 。 TRUE 值表示 PeekMessage 已经检索到了一条消息，该消息可以通过 TranslateMessage 和 DispatchMessage 发送到程序的窗口过程。若程序的消息队列中没有消息，Windows 就允许其它程序处理其消息队列中的消息。若所有消息队列中均无任何消息，Windows 就将控制传给正在调用 PeekMessage 的程序（即 AbortProc 函数）。因而 PeekMessage 的返回值为 FALSE ， AbortProc 就将控制返回给 Windows 。

15.4.3 Windows 如何使用 AbortProc

当程序进行打印时，大部分工作发生在调用 EndPage 的过程中。调用 EndPage 前，程序每调用一次GDI绘图函数，GDI模块只是简单地将另一个记录加到磁盘上的增强型元文件中。当 GDI 得到 EndPage 后，对打印页中由设备驱动程序定义的每个带，GDI 都将该元文件送入设备驱动程序中。然后，GDI 将打印机驱动程序创建的打印输出保存到一个文件中。如果没有启用后台打印，那么 GDI 模块必须自己动将该打印输出写入打印机。

在 EndPage 调用期间，GDI 模块调用你设置的放弃过程。通常 iCode 参数为零，但如果由于存在未打印的其它临时文件而引起 GDI 运行时磁盘空间不够，iCode 参数就为 SP_OUTOFDISK（通常你不检查这个值，但是如果愿意，你可以进行检查）。放弃过程随后进入 PeekMessage 循环从自己的消息队列中检索消息。

如果在程序的消息队列中没有消息，PeekMessage 返回 FALSE，然后放弃过程跳出它的消息循环并给 GDI 模块返回一个 TRUE 值，指示打印应该继续进行。然后 GDI 模块继续处理 EndPage 调用。

如果有错误发生，那么 GDI 将中止打印过程，这样，放弃过程的主要目的是允许用户取消打印。为此，我们还需要一个显示 Cancel 按钮的对话框，让我们采用两个独立的步骤。首先，我们在创建 PRINT2 程序时增加一个放弃过程，然后在 PRINT3 中增加一个带有 Cancel 按钮的对话框，使放弃过程可用。

15.4.4 实现放弃过程

现在快速复习一下放弃过程的机制。可以定义一个如下所示的放弃过程：

```
BOOL CALLBACK AbortProc (HDC hdcPrn, int iCode)
    {
    MSG msg ;
    while (peekMessage (&msg, NULL, 0, 0, PM_REMOVE))
        {
        TranslateMessage (&msg) ;
        DispatchMessage (&msy) ;
        }
    return TRUE:
    }
```

当你想打印什么时，使用下面的调用将 AbortProc 指针传给 Windows:

```
SetAbortProc (hdcPrn, AbortProc) ;
```

在调用 StartDoc 之前进行这个调用就行了。

不过，事情没这么简单。我们忽视了 AbortProc 过程中 PeekMessage 循环这个问题，它是一个很大的问题。仅当程序处于打印过程时，才调用 AbourtProc 过程。如果在 AbortProc 中检索到一条消息并把它传送给窗口过程，那么就会发生某些非常令人讨厌的事情：用户可以从菜单中再次选择 Print，但程序已经处于打印例程之中。当程序在打印前一个文件的

同时，用户也可以把一个新文件装入到程序里。用户甚至可以退出程序！如果这种情况发生了，所有用户程序的窗口都将被清除。最终将从打印例程中返回，但是除了退到不再有效的窗口例程外，你无处可去。

这种东西会把人搞得晕头转向，而我们的程序对此并无准备。正是由于这个原因，当设置放弃过程时，首先应禁用程序窗口，使它不能接受键盘和鼠标输入。可以用以下函数完成这项工作。

```
EnableWindow (hwnd, FALSE) ;
```

它可以禁止键盘和鼠标的输入进入消息队列。因此在打印过程中，用户不能对程序做任何工作。当打印完成时，应重新允许窗口接受输入。

```
EnableWindow (hwnd, TRUE) ;
```

你可能要问，既然没有键盘或鼠标消息进入消息队列，为什么我们还要进行 AbortProc 中的 TranslateMessage 和 DispachMessage 调用呢？实际上并不一定非得需要 TranslateMessage，但是，我们必须使用 DispatchMessage，以防 WM_PAINT 消息进入消息队列中。如果 WM_PAINT 消息没有得到窗口过程中的 BeginPaint 和 EndPaint 的适当处理，由于 PeekMessage 不再返回 FALSE，该消息就会滞留在队列中并且防碍工作。

当在打印期间禁用窗口时，你的程序不会进行显示输出。但用户可以切换到其它程序，并在那里进行其它工作，而后台打印程序则能继续将输出文件送到打印机。

图 15-7 所示的 PRINT2 程序在 PRINT1 的逻辑中增加了一个放弃过程(和必要的支持)。具体地说，PRINT2 增加了一个放弃过程、一个对 SetAbortProc 函数的调用和两个对 EnableWindow 的调用，其中第一个是用来禁止窗口，而第二个是用来启用它。

PRINT2.MAK

```
#----------------------
# PRINT2.MAK make file
#----------------------

print2.exe : print.obj print2.obj
     $(LINKER) $(GUIFLAGS) -OUT:print2.exe print.obj print2.obj \
     $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print2.obj : print2.c
     $(CC) $(CFLAGS) print2.c
```

PRINT2.C

```
/*------------------------------------------
   PRINT2.C -- Printing with Abort Function
               (c) Charles Petzold, 1996
  ------------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC (void) ;              // in PRINT.C
```

```
void PageGDICalls (HDC, int, int) ;

HINSTANCE hInst ;
char       szAppName[] = "Print2" ;
char       szCaption[] = "Print Program 2 (Abort Function)" ;

BOOL CALLBACK AbortProc (HDC hdcPrn, int iCode)
     {
     MSG msg ;

     while (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return TRUE ;
     }

BOOL PrintMyPage (HWND hwnd)
     {
     static DOCINFO di = { sizeof (DOCINFO), "Print2: Printing", NULL } ;
     BOOL           bError = FALSE ;
     HDC            hdcPrn ;
     short          xPage, yPage ;

     if (NULL == (hdcPrn = GetPrinterDC ()))
          return TRUE ;

     xPage = GetDeviceCaps (hdcPrn, HORZRES) ;
     yPage = GetDeviceCaps (hdcPrn, VERTRES) ;

     EnableWindow (hwnd, FALSE) ;

     SetAbortProc (hdcPrn, AbortProc) ;

     if (StartDoc (hdcPrn, &di) > 0)
          {
          if (StartPage (hdcPrn) > 0)
               {
               PageGDICalls (hdcPrn, xPage, yPage) ;

               if (EndPage (hdcPrn) > 0)
                    bError = TRUE ;
               }
          }
     else
          bError = TRUE ;

     if (!bError)
          EndDoc (hdcPrn) ;

     EnableWindow (hwnd, TRUE) ;
     DeleteDC (hdcPrn) ;
     return bError ;
     }
```

图 15-7 PRINT2程序

15.4.5 增加打印对话框

PRINT2 还不能令人十分满意。首先，这个程序没有直接指示出何时开始打印和何时结束打印。只有将鼠标指向程序并且发现它没有反应时，才能断定它仍然在处理 PrintMyPage 例程。PRINT2 在进行后台处理时也没有给用户提供取消打印作业的机会。

你可能注意到，大多数 Windows 程序都为用户提供了一个取消当前正在进行的打印操作的机会。一个小的对话框出现在屏幕上，它包括一些文本和 Cancel 压入按钮。在 GDI 将打印输出保存到磁盘文件或打印机正在打印的整个期间，程序都显示这个对话框。它是一个非模态对话框，你必须提供对话过程。

通常称这个对话框为“放弃对话框”，称这种对话过程为“放弃对话过程”。为了更清楚地把它和“放弃过程”区别开来，我们称这种对话过程为“打印对话过程”。放弃过程 AbortProc 和打印对话过程(将命名为 PrintDlgProc)是两个不同的输出函数。如果想以一种很专业的 Windows 方式进行打印，就必须拥有这两个函数。

这两个函数的交互作用方式如下：必须对 AbortProc 中的 PeekMessage 循环进行修改，以便将消息发送给非模态对话框的对话框窗口过程。PrintDlgProc 必须处理 WM_COMMAND 消息，以检查 Cancel 按钮的状态。如果 Cancel 钮被按下，就将一个叫做 bUserAbort 的变量置为 TRUE。AbortProc 返回的值正好和 bUserAbort 相反。你可能还记得，如果 AbortProc 返回 FALSE 则放弃打印，如果返回 TRUE 则继续打印。在 PRINT2 中，我们总是返回 TRUE。现在，用户在打印对话框中按下 Cancel 按钮时将返回 FALSE，图 15-8 所示的 PRINT3 程序中实现了这个逻辑。

如果你使用 PRINT3，那么最好临时禁用后台打印；否则，只有在后台程序从 PRINT3 中接收数据时才可见的 Cancel 按钮可能会很快消失，让你根本没有机会去按它。为了禁用后台打印，从开始菜单上选择设置，在缺省打印机的图标上点鼠标右键，并选属性。你将在“详细资料”中找到后台设置。

如果当你按 Cancel 按钮时打印并不立即终止（特别是在一个慢速打印机上），不要惊奇。打印机有一个内部缓冲区，在打印机停止之前其中的数据必须全部送出，按 Cancel 只是告诉 GDI 不要向打印机的缓冲区发送更多的数据而已。

PRINT3.MAK

```
#----------------------
# PRINT3.MAK make file
#----------------------

print3.exe : print.obj print3.obj print.res
     $(LINKER) $(GUIFLAGS) -OUT:print3.exe print.obj print3.obj \
     print.res $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print3.obj : print3.c
     $(CC) $(CFLAGS) print3.c
```

```
print.res : print.rc
     $(RC) $(RCVARS) print.rc
```

PRINT3.C

```
/*---------------------------------------
   PRINT3.C -- Printing with Dialog Box
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC (void) ;              // in PRINT.C
void PageGDICalls (HDC, int, int) ;

HINSTANCE hInst ;
char      szAppName[] = "Print3" ;
char      szCaption[] = "Print Program 3 (Dialog Box)" ;

BOOL   bUserAbort ;
HWND   hDlgPrint ;

BOOL CALLBACK PrintDlgProc (HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               SetWindowText (hDlg, szAppName) ;
               EnableMenuItem (GetSystemMenu (hDlg, FALSE), SC_CLOSE,
                                                            MF_GRAYED) ;
               return TRUE ;

          case WM_COMMAND :
               bUserAbort = TRUE ;
               EnableWindow (GetParent (hDlg), TRUE) ;
               DestroyWindow (hDlg) ;
               hDlgPrint = 0 ;
               return TRUE ;
          }
     return FALSE ;
     }

BOOL CALLBACK AbortProc (HDC hdcPrn, int iCode)
     {
     MSG   msg ;

     while (!bUserAbort && PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (!hDlgPrint || !IsDialogMessage (hDlgPrint, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          }
     return !bUserAbort ;
     }
```

```
BOOL PrintMyPage (HWND hwnd)
     {
     static DOCINFO di = { sizeof (DOCINFO), "Print3: Printing", NULL } ;
     BOOL           bError = FALSE ;
     HDC            hdcPrn ;
     int            xPage, yPage ;

     if (NULL == (hdcPrn = GetPrinterDC ()))
          return TRUE ;

     xPage = GetDeviceCaps (hdcPrn, HORZRES) ;
     yPage = GetDeviceCaps (hdcPrn, VERTRES) ;

     EnableWindow (hwnd, FALSE) ;

     bUserAbort = FALSE ;
     hDlgPrint = CreateDialog (hInst, "PrintDlgBox", hwnd, PrintDlgProc) ;

     SetAbortProc (hdcPrn, AbortProc) ;

     if (StartDoc (hdcPrn, &di) > 0)
          {
          if (StartPage (hdcPrn) > 0)
               {
               PageGDICalls (hdcPrn, xPage, yPage) ;

               if (EndPage (hdcPrn) > 0)
                    EndDoc (hdcPrn) ;
               else
                    bError = TRUE ;
               }
          }
     else
          bError = TRUE ;

     if (!bUserAbort)
          {
          EnableWindow (hwnd, TRUE) ;
          DestroyWindow (hDlgPrint) ;
          }

     DeleteDC (hdcPrn) ;

     return bError || bUserAbort ;
     }
```

PRINT.RC

```
/*---------------------------
   PRINT.RC resource script
  ---------------------------*/

#include <windows.h>

PrintDlgBox DIALOG 40, 40, 120, 40
     STYLE WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_VISIBLE
```

```
{
CTEXT         "Cancel Printing", -1,  4,  6, 120, 12
DEFPUSHBUTTON "Cancel",     IDCANCEL, 44, 22,  32, 14, WS_GROUP
}
```

图 15-8 PRINT3程序

PRINT3 增加了两个全局变量：一个是叫做 bUserAbort 的布尔变量，另一个是叫做 hDlgPrint 的对话框窗口句柄。PrintMyPage 函数将 bUserAbort 初始化为 FALSE。与 PRINT2 中一样，程序的主窗口是禁用的。指向 AbortProc 的指针用于 SetAbortProc 调用中，而指向 PrintDlgProc 的指针用于 CreateDialog 调用中。 CreatDialog 返回的窗口句柄保存在 hDlgPrint 中。

现在， AbrtProc 中的消息循环如下：

```
while (!bUserAbort && PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
    {
    if (!hDlgprit || !IsDialogMessage (hDlgPrint,&msg))
        {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
        }
    }
return !bUserAbort;
```

仅当 bUserAbort 为 FALSE，也就是用户还没有终止打印工作时，才调用 PeekMessage。IsDIalogMessage 函数用来将消息发送给非模态对话框。和普通的非模态对话框一样，对话框窗口的句柄在这个调用之前受到检查。 AbortProc 的返回值正好与 bUserAbort 相反。开始时， bUsertAbort 为 FALSE，因此 AbortProc 返回 TRUE，表示继续进行打印；但是 bUserAbort 可能在打印对话过程中被设置为 TRUE。

PrintDlgDroc 函数是相当简单的。当处理 WM_INITDLALOG 时，该函数将窗口标题设置为程序名，并且禁用系统菜单上的 Close 选项。 如果用户按下了 Cancel 按钮，PrintDlgProc 将收到 WM_COMMAND 消息：

```
case WM_COMMAND :
    bUserAbort = TRUE ;
    Enablewindow (Getparent (hDlg), true) ;
    DestroyWindow (hDlg) ;
    hDlgPrint = 0 ;
return TRUE ;
```

将 bUserAbort 设置为 TRUE，则说明用户已经决定取消打印操作，主窗口被启动，而对话框被清除(按顺序完成这两项活动是很重要的，否则，运行在 Windows 中的其它一些程序将变成活动程序，而你的程序将消失到背景中)。与通常情况一样，将 hDlgPrint 设置为 0，防止在消息循环中调用 IsDialogMessage。

仅当 AbortProc 用 PeekMessage 检索到消息并用 IsDialogMessage 将它们传送给对话框窗口过程时，这个对话框才接收消息。仅当 GDI 模块处理 EndPage 函数时，才调用 AbortProc。如果 GDI 发现 AbortProc 的返回值是 FALSE，它将控制从 EndPage 返回到

PrintMyPage。它会不返回错误码。至此，PrintMyPage 认为页已经发完了，并调用 EndDoc 函数。但是，由于 GDI 模块还没有完成对 EndPage 调用的处理，所以不会打印出什么东西来。

某些清除工作尚待完成。如果用户没在对话框中取消打印作业，那么仍然显示对话框。PrintMyPage 重新启用它的主窗口并清除对话框:

```
if (!bUserAbort)
    {
    EnableWindow (hwnd,TRUE) ;
    DestroyWindow (hDlgPrint) ;
    }
```

两个变量会通知你发生了什么事：bUserAbort 可以告诉你用户是否终止了打印作业，bError 会告诉你是否出了故障，你可以用这些变量来完成想做的工作。PrintMyPage 只简单地对它们进行逻辑或运算，把值返回给 WndProc：

```
return bError bUserAbort
```

15.4.6 为 POPPAD 增加打印功能

现在准备在 POPPAD 程序中增加打印功能，并且宣布 POPPAD 已告完毕。这需要第 11 章中的各种 POPPAD 文件，此外，还需要图 15-9 中的两个新文件。

POPPAD.MAK

```
#-----------------------
# poppad.MAK make file
#-----------------------

poppad.exe : poppad.obj popfile.obj popfind.obj \
             popfont.obj popprnt.obj poppad.res
     $(LINKER) $(GUIFLAGS) -OUT:poppad.exe poppad.obj popfile.obj \
     popfind.obj popfont.obj popprnt.obj poppad.res $(GUILIBS)

poppad.obj : poppad.c poppad.h
     $(CC) $(CFLAGS) poppad.c

popfile.obj : popfile.c
     $(CC) $(CFLAGS) popfile.c

popfind.obj : popfind.c
     $(CC) $(CFLAGS) popfind.c

popfont.obj : popfont.c
     $(CC) $(CFLAGS) popfont.c

popprnt.obj : popprnt.c
     $(CC) $(CFLAGS) popprnt.c

poppad.res : poppad.rc poppad.h poppad.ico
     $(RC) $(RCVARS) poppad.rc
```

POPPRNT.C

```
/*--------------------------------------------
   POPPRNT.C -- Popup Editor Printing Functions
  --------------------------------------------*/

#include <windows.h>
#include <commdlg.h>
#include <string.h>
#include "poppad.h"

BOOL bUserAbort ;
HWND hDlgPrint ;

BOOL CALLBACK PrintDlgProc (HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               EnableMenuItem (GetSystemMenu (hDlg, FALSE), SC_CLOSE,
                                                            MF_GRAYED) ;
               return TRUE ;

          case WM_COMMAND :
               bUserAbort = TRUE ;
               EnableWindow (GetParent (hDlg), TRUE) ;
               DestroyWindow (hDlg) ;
               hDlgPrint = 0 ;
               return TRUE ;
          }
     return FALSE ;
     }

BOOL CALLBACK AbortProc (HDC hPrinterDC, int iCode)
     {
     MSG msg ;

     while (!bUserAbort && PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (!hDlgPrint || !IsDialogMessage (hDlgPrint, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          }
     return !bUserAbort ;
     }

BOOL PopPrntPrintFile (HINSTANCE hInst, HWND hwnd, HWND hwndEdit,
                                                  LPSTR szTitleName)
     {
     static DOCINFO  di = { sizeof (DOCINFO), "", NULL } ;
     static PRINTDLG pd ;
     BOOL            bSuccess ;
     LPCTSTR         pstrBuffer ;
     int             yChar, iCharsPerLine, iLinesPerPage, iTotalLines,
                     iTotalPages, iPage, iLine, iLineNum ;
```

```
     TEXTMETRIC        tm ;
     WORD              iColCopy, iNonColCopy ;

     pd.lStructSize         = sizeof (PRINTDLG) ;
     pd.hwndOwner           = hwnd ;
     pd.hDevMode            = NULL ;
     pd.hDevNames           = NULL ;
     pd.hDC                 = NULL ;
     pd.Flags               = PD_ALLPAGES | PD_COLLATE | PD_RETURNDC ;
     pd.nFromPage           = 0 ;
     pd.nToPage             = 0 ;
     pd.nMinPage            = 0 ;
     pd.nMaxPage            = 0 ;
     pd.nCopies             = 1 ;
     pd.hInstance           = NULL ;
     pd.lCustData           = 0L ;
     pd.lpfnPrintHook       = NULL ;
     pd.lpfnSetupHook       = NULL ;
     pd.lpPrintTemplateName = NULL ;
     pd.lpSetupTemplateName = NULL ;
     pd.hPrintTemplate      = NULL ;
     pd.hSetupTemplate      = NULL ;

     if (!PrintDlg (&pd))
          return TRUE ;

     iTotalLines = (short) SendMessage (hwndEdit, EM_GETLINECOUNT, 0, 0L) ;

     if (iTotalLines == 0)
          return TRUE ;

     GetTextMetrics (pd.hDC, &tm) ;
     yChar = tm.tmHeight + tm.tmExternalLeading ;

     iCharsPerLine = GetDeviceCaps (pd.hDC, HORZRES) / tm.tmAveCharWidth ;
     iLinesPerPage = GetDeviceCaps (pd.hDC, VERTRES) / yChar ;
     iTotalPages   = (iTotalLines + iLinesPerPage - 1) / iLinesPerPage ;

     pstrBuffer = (LPCTSTR) HeapAlloc (GetProcessHeap (),
                                HEAP_NO_SERIALIZE, iCharsPerLine + 1) ;

     EnableWindow (hwnd, FALSE) ;

     bSuccess   = TRUE ;
     bUserAbort = FALSE ;

     hDlgPrint = CreateDialog (hInst, (LPCTSTR) "PrintDlgBox", hwnd,
                                          PrintDlgProc) ;
     SetDlgItemText (hDlgPrint, IDD_FNAME, szTitleName) ;

     SetAbortProc (pd.hDC, AbortProc) ;

     GetWindowText (hwnd, (PTSTR) di.lpszDocName, sizeof (PTSTR)) ;

     if (StartDoc (pd.hDC, &di) > 0)
          {
```

```
          for (iColCopy = 0 ;
               iColCopy < ((WORD) pd.Flags & PD_COLLATE ? pd.nCopies : 1) ;
               iColCopy++)
               {
               for (iPage = 0 ; iPage < iTotalPages ; iPage++)
                    {
                    for (iNonColCopy = 0 ;
                         iNonColCopy < (pd.Flags & PD_COLLATE ?
                                             1 : pd.nCopies) ;
                         iNonColCopy++)
                         {

                         if (StartPage (pd.hDC) < 0)
                              {
                              bSuccess = FALSE ;
                              break ;
                              }

                         for (iLine = 0 ; iLine < iLinesPerPage ; iLine++)
                              {
                              iLineNum = iLinesPerPage * iPage + iLine ;

                              if (iLineNum > iTotalLines)
                                   break ;

                              *(int *) pstrBuffer = iCharsPerLine ;

                              TextOut (pd.hDC, 0, yChar * iLine, pstrBuffer,
                                   (int) SendMessage (hwndEdit, EM_GETLINE,
                                   (WPARAM) iLineNum, (LPARAM) pstrBuffer)) ;
                              }

                         if (EndPage (pd.hDC) < 0)
                              {
                              bSuccess = FALSE ;
                              break ;
                              }

                         if (bUserAbort)
                              break ;
                         }

                    if (!bSuccess || bUserAbort)
                         break ;
                    }

               if (!bSuccess || bUserAbort)
                    break ;
               }
          }
     else
          bSuccess = FALSE ;

     if (bSuccess)
          EndDoc (pd.hDC) ;
```

```
if (!bUserAbort)
     {
     EnableWindow (hwnd, TRUE) ;
     DestroyWindow (hDlgPrint) ;
     }

HeapFree (GetProcessHeap (), 0, (LPVOID) pstrBuffer) ;
DeleteDC (pd.hDC) ;

return bSuccess && !bUserAbort ;
}
```

图 15-9 增加打印功能的新的POPPAD文件

与 POPPAD 尽量利用 Windows 高级特性来简化程序的方针相一致，POPPRNT 文件使用了 PrintDlg 函数。这个函数包括在通用对话框库中，以一个 PRINTDLG 类型的结构为参数。

通常，一个 Print 选项被包含在程序的 File 菜单中。当用户选中 Print 选项时，程序可以初始化 PRINTDLG 结构的域，并调用 PrintDlg 。

PrintDlg 显示一个对话框，它允许用户选择打印页的范围。因此，这个对话框特别适用于像 POPPAD 这样能打印多页文档的程序。这种对话框同时也给出了一个确定拷贝份数的编辑区和名为"归类（Collate）"的复选框。归类影响着多个拷贝页的顺序。例如，如果文档是 3 页，用户要求打印三份拷贝，则这个程序能以两种顺序之一打印它们。归类后的拷贝的页码顺序为 1、2、3、1、2、3、1、2、3，未经归类的拷贝的页码顺序是 1、1、1、2、2、2、3、3、3。程序的责任是以正确的顺序打印拷贝。

这个对话框也允许用户选择非缺省打印机，它包括一个标记为 Properties 的按钮，可以激活设备方式对话框。这样，至少允许用户选择肖像或风景方式。

从 PrintDlg 函数返回后，PRINTDLG 结构的域指明打印页的范围和是否对多个拷贝进行归类。这个结构同时也给出了准备使用的打印机设备环境句柄。

在 POPPAD.C 中，PopPrntPrintFile 函数(当用户在 File 菜单里选中 Print 选项时，它由 POPPAD 调用)调用 PrintDlg，然后开始打印文件。PopPrntPrintFile 完成某些计算，以确定一行能容纳多少字符和一页能容纳多少行。这个过程涉及到调用 GetDeviceCaps 来确定页的分辨率，调用 GetTextMetrics 来确定字符的大小。

这个程序通过发送一条 EM_GETLINECOUNT 消息给编辑控制来获取文档中的总行数（在变量 iTotalLines 中）。保存各行内容的缓冲区分配在局部内存中。对每一行，缓冲区的第一个字被设置为该行中字符的数目。把 EM_GETLINE 消息发送给编辑控制可以把一行拷贝到缓冲区中，然后用 TextOut 把这一行送到打印机设备环境中。

为了确定拷贝份数，应注意打印文本的逻辑包括两个 for 循环。第一个 for 循环使用了一个叫作 iColCopy 的变量，当用户指定将拷贝归类时，它将起作用。第二个 for 循环使用了一个叫作 iNonColCopy 的变量，当不对拷贝进行归类时，它将起作用。

如果 StartPage 或者 EndPage 返回一个错误或者如果 bUserAbort 为 TRUE，那么这个程序退出增加页号的那个 for 循环。如果放弃过程的返回值是 FALSE，则 EndPage 不返回错误。正是由于这个原因，在下一页开始之前，要直接测试 bAUserAbort 。如果没有报告错误，则进行 EndDoc 调用。

```
if (!bError)
    EndDoc (hdcPrn) ;
```

你可能想通过打印多页文件来测试 POPPAD。你可以从打印任务窗口中监视打印进展情况，通过从打印机文件夹的 File 菜单中选择 Open 来打开这个窗口。在 GDI 处理完第一个 EndPage 调用之后，首先打印的文件将显示在打印任务窗口中。此时，后台打印程序开始把文件发送到打印机。然后，如果在 POPPAD 中取消打印作业，那么后台打印程序将终止打印，这也就是放弃过程返回 FALSE 的结果。当文件出现在打印任务窗口中，你也可以通过从 Document 菜单中选择 Cancel Printing 来取消打印作业，在这种情况下，POPPAD 中的 EndPage 调用返回 SP_ERROR 错误(等于－1)。

Windows 的程序设计的新手经常会抱住 AbortDoc 函数不放，但实际上这个函数几乎不在打印中使用。像在 POPPAD 中看到的那样，用户几乎随时都可以取消打印作业，或者通过 POPPAD 的打印对话框或者通过打印任务窗口。这两种方法都不需要程序使用 AbortDoc 函数。POPPAD 中允许 AbortDoc 的唯一时刻是在对 StartDoc 的调用和对 EndPage 的第一个调用之间，但是程序很快就会执行过去，以至不再需要 AbortProc。

图 15-10 显示出正确打印多页文档的打印函数的调用顺序。检查 bUserAbort 的值是否为 TRUE 的最佳位置是在每个 EndPage 函数之后。实际上，如果任何一个打印函数的调用出现错误，那么表演就结束了，你也可以回家了。

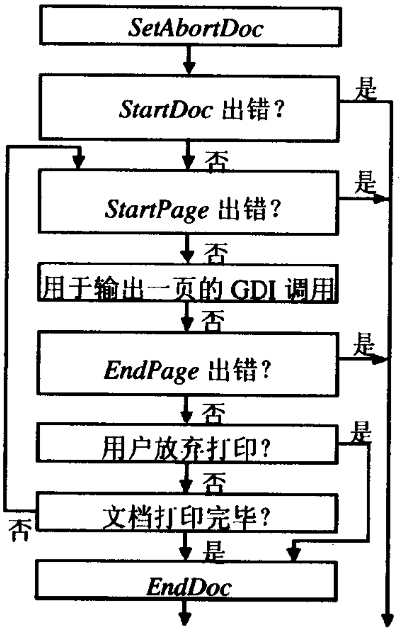


图 15-10　打印一个文档时的函数调用顺序

15.4.7　处理错误码

前面已经用相对简单的方式处理了打印函数的返回值：如果打印函数（如 StartDoc）返回一个负值，则出现了错误，打印工作被终止。负的错误代码是 SP_ERROR(－1)，这也是 Win32 打印函数失败返回的唯一的值。SP_ERROR 本身并不能告诉我们很多东西。

你可以通过在错误发生后立即调用 GetLastError，向用户报告错误的更准确情况。当一个打印函数失败之后调用 GetLastError 时，它返回 WINDOWS.H 中定义的 5 个标识符之一。WINDOWS.H 同时也包括一个叫做 SP_NOTREPORTED 的标识符，它等于 0x4000。如果 GetLastError 的返回值与 SP_NOTREPORTED 按位与后等于 0，那么错误已经报告给了用户。可以把 GetLastError 的返回值与 SP_NOTREPORTED 按位 " OR "，再将结果值与 5 个错误码标识符进行比较，以确定是否已经报告了错误。

下面的代码段给出了一种获取错误的文本字符串说明的方法。它假定如果一个 StartDoc 这样的打印函数返回一个负值，那么 bError 标志就被设置为 TRUE。程序代码检测这个标志并在错误发生时调用函数_GetErrorText，然后它将返回的错误消息显示给用户。如果错误已经报告给用户，该函数返回 NULL。

```
if (bError == TRUE)
     {
     pszErrMsg = _GetErrorText () ;
     if (pszErrMsg)
          MessageBox (hwnd, pszErrMsg, NULL, MB_OK) ;
     }
.
.
LPCTSTR _GetErrorText (void)
     {
     static LPCTSTR pszErrorText[] = { "General error",
                                       "Canceled from program",
                                       "Canceled by user",
                                       "Out of disk space",
                                       "Out of memory" } ;
     DWORD dwError = GetLastError () ;

     if ((dwError & SP_NOTREPORTED) == 0)
          return NULL ;

     return pszErrorText[~dwError] ;
     }
```

五个错误码如下:

- SP_ERROR (OxFFFF 或－1)——指示一个“一般性错误”，这是 Win32 打印函数返回的唯一错误码，它表示 GDI 模块或打印机设备驱动程序不能创建一个文档。
- SP_APPABORT (OxFFFE 或－2)——说明程序的放弃过程已经返回 FALSE 值。但是，仅当不启用后台程序时这种情况才会出现；如果后台程序是活动的，并且放弃过程收到了值为 0 的 iCode 参数并返回了 FALSE，那么，EndPage 调用将返回一个正值。
- SP_USERABORT(OxFFFD 或－3)——说明用户从打印机文件夹取消了打印作业。
- SP_OUTOFDISK (OxFFFC 或－4)——说明没有可用的空闲磁盘空间。如果包含 TEMP 子目录的磁盘驱动器不能容纳任何临时后台文件，就会遇到这个错误码。如果 TEMP 子目录中已经存在一些临时后台文件，那么在 EndPage 期间将以 SP_OUTOFDISK 作为 iCode 参数调用放弃过程。如果放弃过程随后返回 FALSE，那么 EndPage 调用返回 SP_ERROR。接着对 GetLastError 的调用将返回

SP_OUTOFDISK 。

- SP_OUTOFMEMORY(OxFFFB 或－5)——说明没有足够的内存用于打印。

15.5　打印分带技术

打印分带是一种将一个图形页定义为一系列独立的称为带的矩形的技术，这种方法摆脱了打印机驱动程序在内存中构造整个位图图象页的需要。打印分带技术对不具备高级页面制作控制功能的打印机（例如点阵打印机和某些激光打印机）来说是很重要的。

在 Windows 中，打印分带是打印程序设计中最让人误解的一个方面，其中的部分原因在于 GetDeviceCaps 函数的文档说明。GetDeviceCaps 用 RASTERCAPS 索引返回的值的 RC_BANDING 位在文档中被解释成“需要打印分带支持”。看到这项说明的程序员就会假定其应用程序必须针对这样的打印机使用打印分带技术，其实情况并非如此。从 GetDeviceCaps 得到的大多数信息都是对 GDI 模块而言的，这些信息允许 GDI 决定它自己能利用什么设备和需要什么帮助。打印分带需求就属于这一类。

一般说来，一个应用程序不需要包括它自己的打印分带代码。像已经出现过的那样，绘制一幅图形的 GDI 调用存储在一个元文件中，然后在打印机驱动程序利用这个元文件之前，使用打印分带技术设置剪裁区域，这个过程对应用程序而言是透明的。但是在某些情况下，应用程序可能想接管完成打印分带的责任。当一个应用程序使用打印分带时，GDI 模块并不生成中间元文件，而恰恰相反，每个带的绘图命令都要传递到打印机设备驱动程序。这种方法有两个优点：

- 可以提高打印速度。应用程序只需要调用在每个特定的带内绘制某些东西的 GDI 函数，这比为每个带使用 GDI 模块把整个元文件输出到设备驱动程序要快得多。
- 可以减少通常情况下打印所需的磁盘空间。如果应用程序正在打印位图而它自己没有分带，那么这些位图必须存储在 GDI 生成的元文件中。这可能导致元文件与 GDI 模块最终生成的打印机输出文件一样大。

打印分带对打印位图尤为重要，这是由于它们在元文件中占据相当大的空间（打印位图需要把位图选进内存设备环境中，然后用 BitBlt 或 StretchBlt 把它写到打印机设备环境中)。但是，打印分带使打印过程更加复杂了，当讨论我们的最后一个版本的打印程序 PRINT4 时，就会看到这一点。

15.5.1　分带原理

为了使程序完成自己的打印分带工作，你必须使用一个我们尚未讨论过的 GDI 函数，叫做 ExtEscape 。这个函数的名字暗示了这是一个被 GDI 模块忽略而直接与打印机驱动程序通讯的函数。在某些情况下，确实如此，但是 GDI 经常也会在 ExtEscape 调用中做一些工作。

ExtEscape 的一般语法为：

```
iResult = ExtEscape (hdcPrinter, iEscapeCode, cbCountIn,
```

```
                    psDataIn, cbCountOut, psDataOut) ;
```

iEscapeCode 参数是使用在 WINGDI.H 中定义的一个标识符来指定的一个子函数码，其它四个参数依赖于这个子函数。虽然两个参数 psDataIn 和 psDataOut 被定义为字符串的指针，但有时它们也指向结构。要将指针强制为串指针，使用（LPSTR）。

并非所有 ExtEscape 子函数在所有的设备驱动程序中都得以实现。事实上，ExtEscape 的设计是开放的，因此显示设备的制造商可以定义它们自己的 ExtEscape 子函数来访问设备特有的某些功能。如果子函数没有被实现，那么 ExtEscape 总是返回 0 。而如果发生了错误，则将返回一个负值。正的返回值表示成功。

我们在这里唯一关心的 ExtEscape 子函数是 NEXTBAND ，这个子函数被用来打印在一系列矩形带中的打印数据。为了在 ExtEscape 调用中使用 NEXTBAND ，首先要定义一个 RECT 类型的变量:

```
RECT rect;
```

你可以回想到 RECT 结构有 4 个叫做 left 、 top 、 right 和 bottom 的域。对每一页而言，都要从调用 ExtEscape 的子函数 NEXTBAND 开始，并且传给它一个 rect 指针。在返回时，rect 中包含第一个带的坐标。不论打印机设备环境的当前映射模式如何，这个位置总是设备坐标(象素)。用 GDI 调用在这个带中进行打印，然后再一次调用 NEXTBAND ExtEscape 函数获取下一个带的位置，而后再打印这个带。当传递给 ExtEsape 的 RECT 结构返回空(所有的域都被置为 0 ）时，表明此页已被打印完毕。

以下是打印一页的程序行。为了简化起见，这几行程序没有考虑 ExtEscape 函数返回的错误或检查 bUserAbort 的值。

```
StartPage (hdcPrn) ;          // Begin one page of print
ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect), (LPSTR) &rect) ;

while (!IsRectEmpty (&rect))
     {
     [call GDI functions to print in band]
     ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect), (LPSTR) &rect) ;
     }

EndPage (hdcPrn) ;            // End the page
```

每个 NEXTBAND ExtEscape 调用(第一个除外)完成与 EndPage 调用相似的功能：它通知 GDI 模块和打印机设备驱动程序，整个带已绘制完毕，现在可以将其保存到磁盘文件中了(或者如果未启用后台打印则写入打印机)。这段代码以 StartPage 的一个调用开始；在循环运行完之后，又以 EndPage 的一个调用结束。

对点阵打印机而言，打印分带是容易实现的。在说明这个过程之前，需要区别“纸头”（它总是纸的首先打印的那部分）和“页头”（它取决于打印机驱动程序是处于肖像模式还是风景模式）。

在肖像模式下，纸头和页头是一样的，打印带沿页面向下伸展。由 NEXTBAND ExtEscape 调用设置的 RECT 结构中的 rect.left 的值总是为 0 ，rect.right 总是等于以象素为单位的打印区域的宽度(这个值是用 HORZRES 参数调用 GetDeviceCaps 时获得的)。对第一

个带来说, rect.top 等于 0 。对后续的带来说， rect.top 等于前一个带的 rect.bottom 值。对最后一个带来说， rect.bottom 等于以象素为单位的打印区域的高度（见图 15-11 ）。

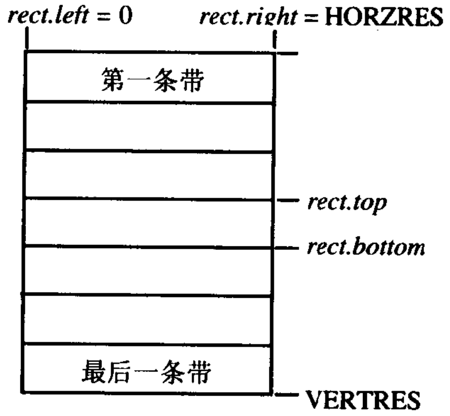


图 15-11 在肖像模式中一个点阵打印机的分带

这样，在每个带中，可以从 rect.left 和 rect.top 位置一直打印到 rect.right 和 rect.bottom 位置。如果调用函数：

```
Rectangle (hdcPrn, rect.left, rect.top, rect.right, rect.bottom) ;
```

矩形将打印在带的最外围（注意，实际上 Rectangle 所绘制的矩形的右边和底边比由上面两个参数说明的尺寸要少一个象素点）。

在风景模式下，点阵打印机必须从页的左边开始打印文档。带在纸上是一样大的，但由于现在页的左边是纸头，所以矩形的坐标不同。在风景模式下， rect.top 总是为 0 ， rect.bottom 恒等于以象素为单位的打印区域高度（这个值是用 VERTRES 参数调用 GetDeviceCaps 时取得的）。 对第一个带而言， rect.right 是以象素为单位的打印区域的宽度（见图 15-12 ）。

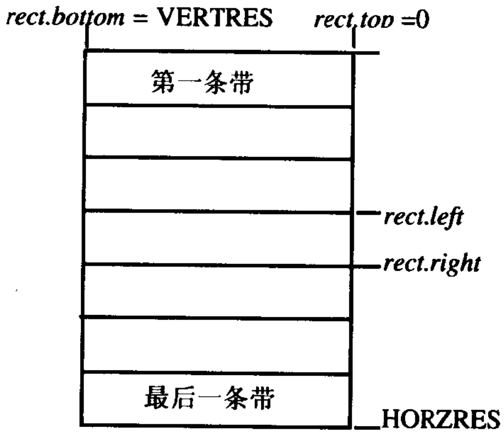


图 15-12 在风景模式中一个点阵打印机的分带

激光打印机或绘图仪处理带的方法可能与点阵打印机不同，这是由于可能不需要把打印机输出从页头到页尾按顺序地输送到打印机上。尽管图 15-11 和图 15-12 代表着一般情况，但你的程序不应该假定打印分带矩形服从这些模式。

将打印机输出分成带看起来是一个很令人头痛的问题，但是即使使用了打印分带，也不需要包含很多打印分带代码。带是一种剪裁区域，你可以执行在带外打印的 GDI 调用，但是，除了落入带中的内容外， Windows 将忽略任何东西。这意味着对每个带来说，都可以进行整个页的 GDI 调用。

可以检查用 RASTERCAPS 参数调用 GetDeviceCaps 时返回的值的 RC_BANDING 位，以便决定某个特定的驱动程序是否需要打印分带支持。正像前面提到的那样，这条信息只与 GDI 有关。不论驱动程序是否需要打印分带支持， GDI 模块总是支持 NEXTBAND ExtEscape 调用的。如果驱动程序不需要打印分带支持，那么一页的第一个 NEXTBAND ExtEscape 调用返回一个大小等于打印区域的矩形，一页的第二个 NEXTBAND 调用返回一个空矩形。

15.5.2 分带技术的实现

如图 15-13 所示的 PRINT4 程序在 PRINT3 中的打印逻辑里增加了分带技术。与前面提到的所有 PRINT 程序一样， PRINT4 同样也需要图 15-8 中的 PRINT.RC 文件和图 15-5 中的 PRINT.C 文件。

PRINT4.MAK

```
#----------------------
# PRINT4.MAK make file
#----------------------

print4.exe : print.obj print4.obj print.res
     $(LINKER) $(GUIFLAGS) -OUT:print4.exe print.obj print4.obj \
     print.res $(GUILIBS) winspool.lib

print.obj : print.c
     $(CC) $(CFLAGS) print.c

print4.obj : print4.c
     $(CC) $(CFLAGS) print4.c

print.res : print.rc
     $(RC) $(RCVARS) print.rc
```

PRINT4.C

```
/*---------------------------------------
   PRINT4.C -- Printing with Banding
               (c) Charles Petzold, 1996
  ---------------------------------------*/

#include <windows.h>

HDC  GetPrinterDC (void) ;              // in PRINT.C
void PageGDICalls (HDC, int, int) ;
```

```
HINSTANCE hInst ;
char      szAppName[] = "Print4" ;
char      szCaption[] = "Print Program 4 (Banding)" ;

BOOL   bUserAbort ;
HWND   hDlgPrint ;

BOOL CALLBACK PrintDlgProc (HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
     {
     switch (msg)
          {
          case WM_INITDIALOG :
               SetWindowText (hDlg, szAppName) ;
               EnableMenuItem (GetSystemMenu (hDlg, FALSE), SC_CLOSE,
                                                            MF_GRAYED) ;
               return TRUE ;

          case WM_COMMAND :
               bUserAbort = TRUE ;
               EnableWindow (GetParent (hDlg), TRUE) ;
               DestroyWindow (hDlg) ;
               hDlgPrint = 0 ;
               return TRUE ;
          }
     return FALSE ;
     }

BOOL CALLBACK AbortProc (HDC hdcPrn, int iCode)
     {
     MSG   msg ;

     while (!bUserAbort && PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (!hDlgPrint || !IsDialogMessage (hDlgPrint, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          }
     return !bUserAbort ;
     }

BOOL PrintMyPage (HWND hwnd)
     {
     static DOCINFO di = { sizeof (DOCINFO), "Print4: Printing", NULL } ;
     BOOL           bError = FALSE ;
     HDC            hdcPrn ;
     RECT           rect ;
     int            xPage, yPage ;

     if (NULL == (hdcPrn = GetPrinterDC ()))
          return TRUE ;

     xPage = GetDeviceCaps (hdcPrn, HORZRES) ;
     yPage = GetDeviceCaps (hdcPrn, VERTRES) ;
```

```
EnableWindow (hwnd, FALSE) ;

bUserAbort = FALSE ;
hDlgPrint = CreateDialog (hInst, "PrintDlgBox", hwnd, PrintDlgProc) ;

SetAbortProc (hdcPrn, AbortProc) ;

if (StartDoc (hdcPrn, &di) > 0 && StartPage (hdcPrn) > 0 &&
    ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect), (LPSTR) &rect)
    > 0)
     {
     while (!IsRectEmpty (&rect) && !bUserAbort)
          {
          Rectangle (hdcPrn, rect.left, rect.top, rect.right,
                                            rect.bottom) ;
          PageGDICalls (hdcPrn, xPage, yPage) ;

          if (ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect),
              (LPSTR) &rect) < 0)
               {
               bError = TRUE ;          // If error, set flag and
               break ;                  //   break out of loop
               }
          }
     }
else
     bError = TRUE ;

if (!bError)
     {
     if (bUserAbort)
          AbortDoc (hdcPrn) ;
     else
          if (EndPage (hdcPrn))
               EndDoc (hdcPrn) ;
     }

if (!bUserAbort)
     {
     EnableWindow (hwnd, TRUE) ;
     DestroyWindow (hDlgPrint) ;
     }

DeleteDC (hdcPrn) ;

return bError || bUserAbort ;
}
```

图 15-13 PRINT4程序

PRINT4 和 PRINT3 的不同之处仅在于几个细节。可以看到，Rectangle 函数打印每个打印分带的矩形而不是整页的边界上的矩形，这将允许你看到特定打印机的带在什么位置。打印操作的结构如下：

```
if (StartDoc (hdcPrn, &di) > 0 && StartPage (hdcPrn) > 0 &&
    ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect), (LPSTR) &rect) > 0)
```

```
     {
     while (!IsRectEmpty (&rect) && !bUserAbort)
          {
          [make GDI calls]
          if (ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect),
              (LPSTR) &rect) < 0)
              {
              bError = TRUE ;          // If error, set flag and
              break ;                  //   break out of loop
              }
          }
     }
else
     bError = TRUE ;
```

仅当矩形非空并且用户没有在对话框中取消打印作业时，带的 while 循环才继续进行。PRINT4 要检查每个 NEXTBAND ExtEscape 调用的返回值,并且在 ExtEsape 返回负值时设置 bError 。如果 ExtEscape 调用均未返回错误，那么打印作业必须通过调用 EndDoc 来结束或通过调用 AbortProc 来放弃。如果用户在 NEXTBAND 循环期间取消打印，那么必须用 AbortProc 调用来终止打印作业。完成这项工作的代码如下：

```
if (StartDoc (hdcPrn, &di) > 0 && StartPage (hdcPrn) > 0 &&
    ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect), (LPSTR) &rect) > 0)
     {
     while (!IsRectEmpty (&rect) && !bUserAbort)
          {
          [make GDI calls]
          if (ExtEscape (hdcPrn, NEXTBAND, 0, NULL, sizeof (rect),
              (LPSTR) &rect) < 0)
              {
              bError = TRUE ;          // If error, set flag and
              break ;                  //   break out of loop
              }
          }
     }
else
     bError = TRUE ;
```

15.6　打印机和字体

第 4 章里有一个使用基于 GDI 的字体来显示格式化文本的程序 JUSTIFY 。在屏幕上使用格式化文本的程序通常也需要打印这个文本。实际上，一般来说，字处理和桌面出版程序只是用显示器来预览打印输出。

TrueType 提供了一种最简单的方法，它可以准确地在屏幕上显示文本被打印时的外观。可以利用一种方法来限制程序只使用 TrueType 字体。如果用 EnumFonts 函数枚举字体，那么你可以检查 iFonType 参数的 TRUETYPE_FONTTYPE 位，然后利用列表框显示其 TRUETYPE_FONTTYPE 位被置位的那些字体。

如果让用户在通用对话框库（像第 4 章中的 JUSTIFY 程序演示的那样)的 ChooseFont

函数给出的对话框中选择字体，可以把 CF_TTONLY 标志放在 CHOOSEFONT 结构的 Flags 域中，以便只列出 TrueType 字体。

但是，限制为 TrueType 字体的选择可能会让那些已经为其激光打印机买了可卸载字体或字体卡的用户不满，这些打印机的字体不会出现在列表框中供这些用户选择。为了在 ChooseFont 生成的对话框中显示打印机字体，必须在 CHOOSEFONT 结构的 Flags 域中包括 CF_PRINTERFONTS 常量。

同时也必须设置 CHOOSEFONT 结构的 bDC 域为打印机的设备环境句柄。随后，列表框中显示的字体只限于内部打印机字体、TrueType 字体和笔划字体（为了在列表框中取消笔划字体，需要使用标志 CF_NOVECTORFONTS)。

如果让用户选择打印机字体，那么在屏幕上显示的东西可能只会接近打印机要打印的内容。例如，如果打印机提供 15 点阵的 Zapf Chancery，就可以近似地用 15 点阵的 TrueType 字体来逼近，但是 TrueType 的字体宽度与 Zapf Chancery 字体不同。对 TrueType 字体来说，甚至可能出现字符宽度的舍入误差，从而导致屏幕显示和打印机打出的字体之间存在差异。

简而言之，如果你正在编写一个显示最终要输出到打印机的格式化文本的程序，可以依靠一些提前的工作来达到目的。以下是一些指导性原则。

当你在屏幕上显示格式化文本时，希望根据最终打印的样子来安排空间。可以用 GetDeviceCaps 和 DeviceCapabilities 调用来确定纸的尺寸和可打印区域的尺寸。例如，如果打印纸为 8.5 英寸宽，并且用户选择左右边界为 1 英寸，然后想用 6.5 英寸的宽度在屏幕上显示文本。第 4 章讨论的“逻辑 twips ”映射模式适用于这种显示。但是有一点值得考虑。如果用户选择打印机所支持的一种 15 点的字体，你或许只能用比如说 14 点的字体来近似它，但是你不能用这种 14 点的显示字体来确定一个打印行上能容纳多少 15 点字体的文本，相反，必须用打印机字体来确定这一点。同样，必须用打字机字体来确定一页能容纳多少行。

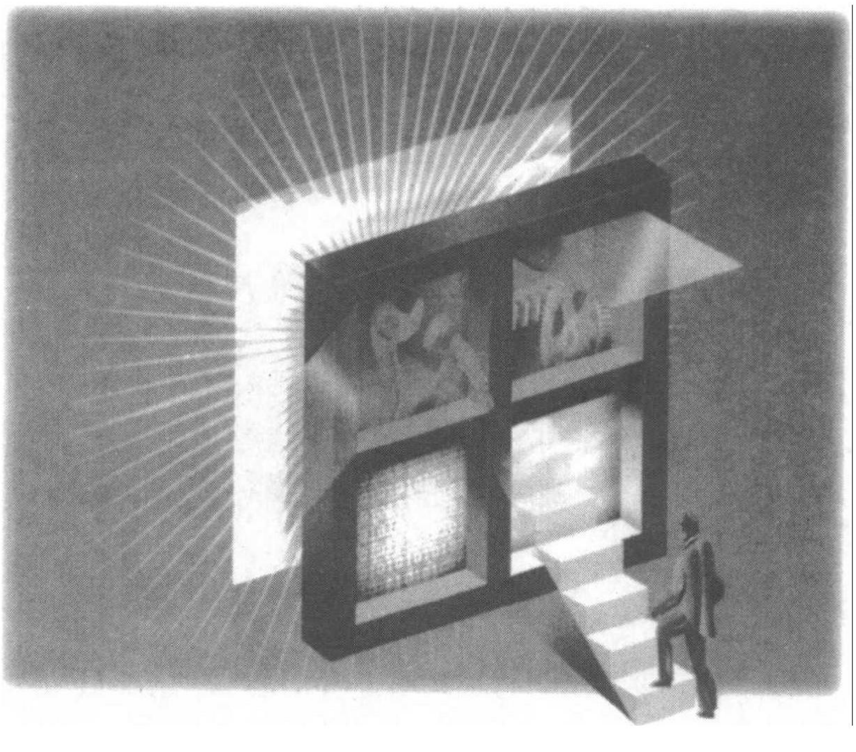
为了格式化显示文本，需要屏幕设备环境句柄（以便在屏幕上显示文本）和打印机信息环境句柄。在实际打印之前，你不会用到打印机设备环境句柄。请遵循下列步骤：

1. 将用户选择的字体名称、大小和属性放到一个逻辑字体结构中，将此逻辑字体选进打印机信息环境中。
2. 为打印机信息环境调用 GetTextMetrics 以便确定选定的打印机字体的实际大小和特征。调用 GetTextFace 获取字体名。
3. 利用第二步获得的信息创建基于打印机字体大小和特征的另一种逻辑字体结构，将这种逻辑字体选进屏幕设备环境中。现在，选进屏幕设备环境的字体很近似选进打印机信息环境中的字体。
4. 当把文本写到屏幕上时，遵循 JUSTIFY 程序中的 Justify 函数所采用的一般过程，只是要将打印机信息环境用于 GetTextExtent 和 SetTextJustification，但不要调用 TextOut。这种方法可确定每行能容纳多少文本和每页能容纳多少行。
5. 当已经建立了适应于打印机的各个文本行后，就可以调用使用屏幕显示环境的 GetTextExtent 和 SetTextJustification 了，然后再调用 TextOut 来显示文本行。

为了打印文本，你可以使用结构与 POPPRINT.C 文件类似的代码并结合 JUSTIFY.C 中 Justify 函数的逻辑。先获取打印机设备环境，再次执行 GetTextExtent 和 SetTextJustification，并用 TextOut 来打印每一行文本。

第五部分

数据交换和链接



第 16 章　剪 贴 板

Windows 剪贴板允许把数据从一个程序传送到另一个程序中。它的原理相对而言比较简单，把数据存放到剪贴板上的程序或从剪贴板上取出数据的程序都无需太多的开销。Windows 95 提供了一个剪贴板浏览程序，该程序可以显示剪贴板的当前内容。许多处理文件或者其它数据的程序都包含一个 Edit 菜单，其中包括 Cut 、 Copy 和 Paste 选项。当用户选择 Cut 或者 Copy 时，程序将数据传送给剪贴板。这个数据使用某种格式，如文本、位图或者元文件等。当用户从菜单中选择 Paste 时，程序检查剪贴板中包含的数据使用的是否是程序可以接受的一种格式，如果是，那么数据将从剪贴板传送到程序中。

如果用户不发出明确的指令，程序就不能把数据送入或移出剪贴板。例如，在某个程序中执行 Cut 或 Copy 操作的用户，应该能够假定数据将保持在剪贴板上，直到下次 Cut 或 Copy 操作为止。

16.1　剪贴板的简单使用

我们从分析把数据传送到剪贴板(Cut 或 Copy)和访问剪贴板数据(Paste)的代码开始。

16.1.1　标准剪贴板数据格式

Windows 支持在 WINDOWS.H 有相应标识符的各种标准剪贴板数据格式。它们是：

- CF_TEXT——以 NULL 结尾的 ANSI 字符集字符串，它在每行末尾包含一个回车和换行符，这是最简单的剪贴板数据格式。传送到剪贴板的数据存放在全局内存块中，并且是利用内存块句柄进行传送的。这个内存块专供剪贴板使用，创建它的程序不应继续使用它。
- CF_BITMAP——与设备有关的位图格式。位图是通过位图句柄传送给剪贴板的。同样，在把这个位图传给剪贴板之后，程序不应再继续使用这个位图。
- CF_METAFLLEPICT——“元文件图片”。它与元文件（第 1 章讨论了元文件）不完全相同，而是一种增加了一些包含在一个 METAFILEPIC 结构中的附加信息的元文件。程序用包含这种结构的全局内存块的句柄把元文件图片传给剪贴板。METAFLLEPICT 结构的四个域是 mm(LONG)——元文件的映射模式；xExt(LONG)和 yExt(LONG)，简单地说，它们就是元文件图象的宽度和高度；bMF(HANDLE)，元文件句柄。在本章的后面将详细讨论 xExt 和 yExt 域。在程序把元文件图片传送给剪贴板后，就不能再继续使用包含 METAFILEPICT 结构的

全局内存块和元文件句柄了，因为它们都处在了 USER 模块的控制之下。

- CF_ENHMETAFILE——增强型元文件的句柄（在第 4 章中描述）。这种格式不使用 METAFILEPICT 结构。
- CF_SYLK——包含 Microsoft “符号链接（ SYLK ）”数据格式的全局内存块。这种格式用在 Microsoft 公司的 Multiplan 、 Chart 和 Excel 程序之间交换数据，它是一种 ASCII 码格式，每一行都用回车和换行结尾。
- CF_DIF——包含数据交换格式（ DIF ）的数据的全局内存块。这种格式是由 Software Arts 公司提出的，用于把数据传送到 VisiCalc 电子表格程序中。 现在 Lotus 公司在控制这种格式。这也是一种 ASCII 码格式，每一行都用回车和换行结尾。

CF_SYLK 和 CF_DIF 格式在概念上类似于 CF_TEXT 格式。但是，包含 SYLK 或者 DIF 数据的字符串不一定要以 NULL 结束，因为这些格式定义了数据的结束。

- CF_TIFF——含有标号图象文件格式（ TIFF ）数据的全局内存块。这种格式由 Microsoft 公司、 Aldus 公司和 Hewlett-Packard 公司以及一些硬件厂商推荐使用。这一格式（它描述了位图数据）可从 Hewlett-Packard 公司获得。
- CF_OEMTEXT——含有文本数据（与 CF_TEXT 类似）的内存块，但是它使用的是 OEM 字符集。
- CF_DIB——定义一个设备无关位图（在第 4 章中描述）的内存块。这种内存块是以 BITMAPINFO 结构开始的，后跟位图数据位。
- CF_PALETTE——调色板句柄。它通常与 CF_DIB 配合使用，以定义位图所使用的颜色调色板。

16.1.2 传送文本到剪贴板

假设想把字符串传送到剪贴板上，并且我们已经有了指向这个字符串的指针(pString)，现在希望传送这个字符串的 iLength 个字节。

首先，通过使用 GlobalAlloc 分配大小为 iLength 个字节的可移动全局内存块，其中还包括一个终止符 NULL:

```
hGlobalMemory = GlobalAlloc (GHND, iLength + 1) ;
```

如果未能分配到内存块， hGlobalMemory 的值将为 NULL 。如果分配成功，则锁定这个块，并得到指向它的一个指针:

```
pGlobalMemory = GlobalLock (hGlobalMemory) ;
```

将字符串拷贝到内存块中:

```
for ( i = 0; i < iLength; i++)
    *pGlobalMemory++ = *pString++ ;
```

由于 GlobalAlloc 的 GHND 标志已使整个内存块在分配期间被清零，所以不需要增加终止 NULL 。以下语句为内存块解锁：

```
GlobalUnlock (hGlobalMemory) ;
```

现在就有了引用以 NULL 结尾的文本所在内存块的内存句柄。为了把它送到剪贴板中，打开剪贴板并把它清空：

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
```

利用 CF_TEXT 标识符把内存句柄交给剪贴板，关闭剪贴板：

```
SetClipboardData (CF_TEXT, hGlobalMemory) ;
CloseClipboard ( ) ;
```

工作告一段落。

下面是关于此过程的一些规则：

- 在处理同一条消息的过程中调用 OpenClipboard 和 CloseClipboard 。当退出窗口过程时，不要让剪贴板打开着。在打开剪贴板时，不要让剪贴板打开过长的时间。
- 不要把锁定的内存句柄交给剪贴板。
- 当调用 SetClipboardData 后，不要再继续使用该全局内存块，它不再属于用户程序，必须把句柄看成是无效的。如果需要继续访问数据，可以制作数据的另一份拷贝，或从剪贴板中读取它（如下节所述）。你也可以在 SetClipboardData 调用和 CloseClipboard 调用之间继续引用内存块，但是你必须使用由 SetClipboardData 返回的句柄。在调用 CloseClipboard 之前，应先为此句柄解锁。

16.1.3 从剪贴板上获取文本

从剪贴板上获取文本只比把文本传送到剪贴板上稍微复杂一些。你必须首先确定剪贴板是否含有 CF_TEXT 格式的数据，最简单的方法是调用：

```
bAvailable = IsClipboardFormatAvailable (CF_TEXT) ;
```

如果剪贴板上含有 CF_TEXT 数据，这个函数将返回 TRUE(非零)。我们在第 10 章的 POPPAD2 程序中已使用了这个函数，用它来确定 Edit 菜单中 Paste 项是被启用的还是被灰化的。 IsClipboardFormatAvailable 是少数几个不需首先打开剪贴板就可以使用的剪贴板函数之一。但是，如果你以后再打开剪贴板得到这个文本，应该再做一次检查(使用同样的函数或其它方法)，以便确定 CF_TEXT 数据是否仍然留在剪贴板中。

为了传送出文本，首先打开剪贴板：

```
OpenClipboard (hwnd) ;
```

得到引用文本的内存块句柄:

```
hClipMemory = GetClipboardData (CF_TEXT) ;
```

如果剪贴板不包含 CF_TEXT 格式的数据，此句柄就为 NULL 。这是确定剪贴板是否含有文本的另一种方法。如果 GetClipboardData 返回 NULL，则关闭剪贴板，不能做其它任何工作。

从 GetClipboardData 得到的句柄并不属于用户程序——它属于剪贴板。仅在 GetClipboardDate 和 CloseClipboard 调用之间这个句柄才有效。你不能释放这个句柄或更改它所引用的数据。如果需要继续访问这些数据，必须制作这个内存块的一份拷贝。

这里有一种将数据拷贝到用户程序中的方法。首先，分配一块与 hClipMemory 所引用的内存块同样大小的内存块:

```
pMyCopy = (char*)malloc (GlobalSize (hClipMemory)) ;
```

现在锁定句柄，获取剪贴板的一个指针：

```
pClipMemory = GlobalLock (hClipMemory) ;
```

现在就可以拷贝数据了：

```
strcpy (pMyCopy, pClipMemory) ;
```

或者，你可以使用一些简单的 C 代码：

```
while (*pMyCopy++ = *pClipMemory++) ;
```

在关闭剪贴板之前先解锁内存块：

```
GlobalUnlock (hClipMemory) ;
CloseClipboard ( ) ;
```

现在你拥有了一个叫做 pMyCopy 的指针，以后可以锁定它来访问这些数据。

16.1.4 打开和关闭剪贴板

在任何时候，只有一个程序可以打开剪贴板。 OpenClipboard 的作用是当一个程序使用剪贴板时，防止剪贴板的内容发生变化。OpenClipboard 返回 BOOL 值，它说明是否已经成功地打开了剪贴板。如果另一个应用程序没有关闭剪贴板，那么它就不能被打开。如果每个程序在响应用户的命令时都尽快地、有礼貌地打开然后关闭剪贴板，那么你将永远不会遇到不能打开剪贴板的问题。

但是，在不礼貌程序和抢占式多任务环境中，有些问题是存在的。即使在你的程序将某些东西放入剪贴板和用户激活一个 Paste 选项期间，你的程序并没有丢失输入焦点，但是也不要假定你放入的东西仍然在那里，一个后台进程可能在此期间访问过剪贴板。

而且，留意与消息框有关的一个更微妙的问题：如果不能分配足够的内存来拷贝剪贴板的内容，那么你可能希望显示一个消息框。但是，如果这个消息框不是系统模态的，那么用户可以在显示消息框期间切换到另一个应用中。你应该或者使消息框是系统模态的，或者在你显示消息框之前关闭剪贴板。

如果你在显示一个对话框时将剪贴板保持为打开状态，那么你还可能会遇到其它问题，对话框中的编辑域会使用剪贴板进行文本的剪切和粘贴。

16.1.5 使用位图剪贴板

在使用 CF_BITMAP 格式时，SetClipboardData 调用交给剪贴板一个设备相关的位图句柄， GetClipboard 函数返回一个设备相关位图的句柄。

如果你拥有一个位图句柄，那么可以使用下面的简单代码将位图拷贝到剪贴板中：

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
SetClipboardData (CF_BITMAP,  hBitmap) ;
CloseClipboard ( ) ;
```

访问该位图也是容易的：

```
OpenClipboard (hwnd) ;
hBitmap = GetClipboardData (CF_BITMAP) ;
```

如果剪贴板中没有包含一个位图，那么 hBitmap 将为 NULL 。为了对位图进行拷贝，你可以调用 GetObject 来确定位图象素大小和颜色组织：

```
GetObject (hBitmap, sizeof (BITMAP), (PSTR)&bm) ;
```

现在，你可以使用 CreateBitmapIndirect ，创建你自己的具有同样大小和颜色组织的位图。

```
hMyBitmap = CreateBitmapIndirect (&bm) ;
```

使用 CreateCompatibleDC 将两个位图选入所创建的内存设备环境中：

```
hdcMemSrc = CreateCompatibleDC (hdc) ;
SelectObject = (hdcMemSrc, hBitmap) ;
hdcMemDst = CreateCompatibleDC (hdc) ;
SelectObject (hdcMemDst, hMyBitmap) ;
```

并调用 BitBlt 将剪贴板位图拷贝为你自己的：

```
BitBlt (hdcMemDst, 0, 0, bm.bmWidth, bm.bmHeight, hdcMemSrc, 0, 0, SRCCOPY) ;
```

仅有的清除工作涉及到删除内存设备环境和关闭剪贴板：

```
DeleteDC (hdcMemSrc) ;
DeleteDC (hdcMemDst) ;
CloaseClipboard ( ) ;
```

不要删除原来的位图，因为现在它属于剪贴板。

使用设备无关位图（ DIB ）的程序还可以将这些位图传送给剪贴板，或者从剪贴板中

获取位图。如果你的程序有一个全局内存句柄，该句柄引用包含一个以 DIB 格式定义的位图的内存块，那么程序可以简单地将句柄传递给使用 CF_DIB 标识符的 SetClipboardData。在关闭剪贴板之后，DIB 内存块不再属于程序。如果程序需要保留一份 DIB 拷贝，可以通过分配同样大小的内存块并拷贝 DIB 数据来为剪贴板制作一份拷贝。从剪贴板上获取 DIB 与从剪贴板上获取文本是一样的，但其数据不是以 0 结尾的。

如果你的程序需要将一个拷贝传送给剪贴板之后保留它自己的一份拷贝（以设备无关或者设备有关的格式），那么你可能会使用许多内存来存储这两个拷贝。在这种情况下，你可能希望使用一种叫做“延迟生成”的技术，以避免浪费内存资源。延迟生成将在本章后面讨论。

16.1.6 元文件和元文件图片

对增强型元文件使用剪贴板是相当简单的。GetClipboardData 函数返回增强型元文件的一个句柄；SetClipboardData 函数也使用元文件句柄。如果需要元文件的一份拷贝，那么可以使用 CopyEnhMetaFile 函数。我在第 4 章已经提到过，并且在第 15 章中再次演示了，增强型元文件格式在老式格式上主要的改进是你可以方便地确定图象的尺寸并在你的窗口中适当地显示图象。

老式的非增强型元文件存在许多问题。如果你有一个老式元文件的句柄，那么当你要使用它时，你将如何确认图象的大小呢？除非你钻到元文件的内部，否则无法做到。

进一步说，当一个程序从剪贴板上获取元文件时，如果元文件被设计成可在 MM_ISOTROPIC 与 MM_ANISOTROPIC 两种映射模式下显示，那么用它来工作就有很大的灵活性。此后，在显示元文件之前，接收元文件的程序就可以通过简单地设置视口范围来缩放图象。但是，如果在元文件里设置映射模式为 MM_ISOTROPIC 或 MM_ANISOTROPIC，那么接收元文件的程序就为难了。程序只有在使用元文件之前或之后才能进行 GDI 调用，在显示元文件的过程中是不能进行 GDI 调用的。

为了解决这些问题，不能把元文件句柄直接送到剪贴板上并由其它程序来调用它。相反，元文件句柄是“元文件图片”的一部分，元文件图片是 METAFILEPICT 类型的结构。这种结构允许从剪贴板上获取元文件图片的程序在使用元文件之前由它自己来设置映射模式和视口范围。

METAFILEPICT 结构为 16 字节长，有四个域：mm(映射模式）、xExt 和 yExt(元文件图象的宽度和高度）、hMF(元文件句柄）。对于除 MM_ISOTROPIC 和 MM_ANISOTROPIC 以外的所有映射模式而言，xExt 和 yExt 值是以 mm 所给定的映射模式中的单位为单位的图象大小。有了这条信息，从剪贴板上拷贝元文件图片结构的程序就能确定执行元文件时要使用多大的显示空间。生成元文件的程序可以把这些值设置成进入元文件的 GDI 绘图函数所使用的最大 x 坐标和 y 坐标。

对 MM_ISOTROPIC 和 MM_ANSOTROPIC 映射模式而言，xExt 和 yExt 的功能不同。你可能回想起第 4 章中的一个程序，当它想在 GDI 函数中使用与图象的可测量尺寸无关的任意逻辑单位时，这个程序使用了 MM_ISOTROPIC 或 MM_ANISOTROPIC 映射模式。如果程序希望在视口尺寸变化时保持纵横比，就使用 MM_ISOTROPIC 映射模式；当它不关

心纵横比时，则使用 MM_ANISOTROPIC 映射模式。你可能还记得第 4 章中的程序，它将映射模式设置为 MM_ISOTROPIC 或 MM_ANISOTROPIC 后，一般要调用 SetWindowExt 和 SetViewportExt 。 SetWindowsExt 使用逻辑单位来指定程序绘图时所使用的单位，SetViewportExt 使用基于视口尺寸的设备单位(例如，窗口客户区的尺寸)。

如果程序生成一个用于剪贴板的 MM_ISOTROPIC 或 MM_ANISOTROPIC 元文件，那么，元文件本身应不包括 SetViewportExt 调用，这是由于该调用中的设备单位可能是基于生成元文件的程序的显示表面，而不是基于从剪贴板读取和显示元文件的程序的显示表面。相反， xExt 和 yExt 的值可帮助从剪贴板上获取元文件的程序恰当地设置显示元文件的视口尺寸。但当映射模式为 MM_ISOTROPIC 或 MM_ANISOTROPIC 时，元文件本身含有设置窗口范围的调用。元文件中 GDI 绘画函数的坐标是基于这些窗口范围的。

生成元文件和元文件图片的程序遵循下列规划：

- 要设置 METAFILEPICT 结构的 mm 域来指出映射模式。
- 对不是 MM_ISOTROPIC 和 MM_ANISOTROPIC 的映射模式来说, xExt 和 yExt 设置为图象的宽度和高度,单位是由 mm 域决定的。若在 MM_ISOTROPIC 或 MM_ANISOTROPIC 环境中使用元文件，情况要稍微复杂一些。对 MM_ANISOTROPIC 来说，当程序既不指定图象的尺寸也不指定图象的纵横比时，xExt 和 yExt 的值都为 0 。对 MM_ISOTROPIC 或 MM_ANISOTROPIC 而言，xExt 和 yExt 的值为正数表明建议使用的图象的宽度和高度单位是 0.01mm(MM_HIMTRIC 单位)。对 MM_ISOTROPIC 来说，xExt 和 yExt 的值为负数表明建议使用的是图象的纵横比而不是尺寸。
- 对 ISOTROPIC 和 MM_ANISOTROPIC 映射模式，元文件本身包含着 SetWindowExtEx 和(可能有)SetWindowOrgEx 调用，也就是说，生成元文件的程序在元文件的设备环境中调用这些函数。一般说来，元文件不包含 SetMapMode 、SetViewpordExt 或 SetViewportOrg 调用。
- 元文件必须是基于内存的元文件，而不是基于磁盘的元文件。

下面是一些生成元文件并把它拷贝到剪贴板中去的示例代码。如果元文件使用 MM_ISOTROPIC 或 MM_AMISOTROPIC 映射模式，那么元文件中的第一个调用应该设置窗口范围(窗口的尺寸在其它映模式中是固定的)。不论映射模式如何，同样应设置原点：

```
hdcMeta = CreateMetaFile (NULL) ;
SetWindowsExtEx (hdcMeta, ...) ;
SetWindowswoOrgEx (hdcMeta, ...) ;
```

元文件中绘图函数所用的坐标基于这些窗口范围和窗口原点。当程序使用 GDI 调用在元文件设备环境上绘图之后，关闭元文件，得到元文件句柄：

```
hmf = CloseMetaFile (hdcMeta) ;
```

程序也需要定义一个指向 METAFILEPICT 类型结构的远指针，并为这个结构分配全局内存块：

```
HGLOBAL        hGMem;
LPMETAFILEPICT pMFP;
```

```
    [其它程序行]
hGMem = GlobalAlloc (GHND, sizeof (METAFILEPICT)) ;

pMFP = (LPMETAFILEPICT) GlobalLock (hGMem) ;
```

程序的下一步是设置这一结构的 4 个域:

```
pMFD->MM = MM_... ;
pMFP->xExt = ... ;
pMFP->yExt = ... ;

GlobalUulock (hGMem) ;
```

然后程序把含有元文件图片结构的内存块送到剪贴板:

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
SetClipboardData (CF_METAFILEPICT, hGMem) ;
CloseClipboard () ;
```

在这些调用之后, hGMem 句柄(含有元文件图片结构的内存块)和 hmf 句柄(元文件本身)对生成它们的程序来说就变成无效的了。

下面讨论比较困难的部分。当程序从剪贴板上获得元文件和使用这个元文件时，必须完成下列步骤:

1. 程序用元文件图片结构的 mm 域来设置映射模式。
2. 对不是 MM_ISOTROPIC 或 MM_ANISOTROPIC 的映射模式而言, 程序用 xExt 和 yExt 的值来设置裁剪矩形或简单地确定图象的尺寸。对 MM_ISOTROPIC 和 MM_ANISOTROPIC 映射模式, 程序用 xExt 和 yExt 来设置视口范围。
3. 然后程序执行元文件。

下面是程序代码。首先打开剪贴板，得到元文件图片结构句柄，并锁定它:

```
OpenClipboard (hwnd) ;
hGMem = GetCUpboardData (CF_METAFILEPICT) ;
pMFP = (LPMETAFILEPICT)GlobalLock (HGMem) ;
```

然后存储当前设备环境的属性, 用结构中的 mm 域设置映射模式:

```
SaveDc (hde) ;
SetMappingMode (lpMFP->mm) ;
```

如果映射模式不是 MM_ISOTROPIC, 可以把裁剪矩形设置为 xExt 和 yExt 的值，由于这些值以逻辑单位为单位, 所以必须用 LPtoDP 把坐标转换为裁剪矩形的设备单位; 或者也可以简单地存储这些值，以便知道图象有多大。

就 MM_ISOTROPIC 或 MM_ANISOTROPIC 映射模式, 要用 xExt 和 yExt 来设置视口范围。以下给出了完成这项工作的一个可能的函数，如果从 xExt 和 yExt 得不到需要的尺寸， 那么，这个函数假定 cxClient 和 cyClient 表示元文件显示区域的象素高度和宽度。

```
void PrepareMetaFile (HDC hdc, LPMETAFILEPICT pmfp,
                         int cxClient, int cyClient)
     {
```

```
    int xScale, yScale, iScale ;

    SetMapMode (hdc, pmfp->mm) ;

    if (pmfp->mm == MM_ISOTROPIC || pmfp->mm == MM_ANISOTROPIC)
         {
         if (pmfp->xExt == 0)
              SetViewportExtEx (hdc, cxClient, cyClient, NULL) ;
         else if (pmfp->xExt > 0)
              SetViewportExtEx (hdc,
                   pmfp->xExt * GetDeviceCaps (hdc, HORZRES) /
                                GetDeviceCaps (hdc, HORZSIZE) / 100,
                   pmfp->yExt * GetDeviceCaps (hdc, VERTRES) /
                                GetDeviceCaps (hdc, VERTSIZE) / 100,
                   NULL) ;

         else if (pmfp->xExt < 0)
              {
              xScale = 100 * cxClient * GetDeviceCaps (hdc, HORZSIZE) /
                             GetDeviceCaps (hdc, HORZRES) / -pmfp->xExt ;
              yScale = 100 * cyClient * GetDeviceCaps (hdc, VERTSIZE) /
                             GetDeviceCaps (hdc, VERTRES) / -pmfp->yExt ;
              iScale = min (xScale, yScale) ;

              SetViewportExtEx (hdc,
                   -pmfp->xExt * iScale * GetDeviceCaps (hdc, HORZRES) /
                             GetDeviceCaps (hdc, HORZSIZE) / 100,
                   -pmfp->yExt * iScale * GetDeviceCaps (hdc, VERTRES) /
                             GetDeviceCaps (hdc, VERTSIZE) / 100,
                   NULL) ;
              }
         }
    }
```

这段程序假定 xExt 和 yExt 同时为 0 、大于 0 或小于 0 。如果为 0, 则不提供尺寸或纵横比。视口的范围设置成在其中希望显示元文件的客户区。 xExt 和 yExt 的值为正数时, 图象尺寸的单位为 0.01mm 。 GetDeviceCaps 函数有助于确定每 0.01mm 的象素数目, 并且要将这个值乘以元文件图片结构中的范围值。 xExt 和 yExt 的值为负数时表示使用的是纵横比而不是尺寸。首先根据以毫米为单位的尺寸的纵横比（相应于 cxClient 和 cyClient)计算出 iScale 的值, 然后用这个缩放因子设置以象素为单位的视口范围。

做完这些以后，如果你愿意，可以设置视口原点，运行元文件，并把设备环境恢复到正常状态：

```
PlayMetaFile (lpMfp->hMF) ;
RestoreDc (hdc, -1) ;
```

然后释放内存块, 关闭剪贴板:

```
GlobalUnlock (hGMem) ;
```

```
CloseClipboard ( ) ;
```

16.2 复杂的剪贴板用法

在使用文本和位图时，读者已经看到，在将数据准备好之后，把数据传送到剪贴板上需要4个调用:

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
SetClipboardData (wFormat, hHandle) ;
CloseClipboard ( ) ;
```

访问这些数据需要三个调用:

```
OpenClipboard (hwnd) ;
hHandle = GetClipboardData (wFormat) ;
[其它程序行]
CloseClipboard ( ) ;
```

在 GetClipboardData 和 CloseClipboard 调用之间，可以制作剪贴板数据的拷贝或以其它方式来使用它。很多应用程序都需要采用这种方法，但也可以用更复杂的方式来使用剪贴板。

16.2.1 利用多个数据项

当打开剪贴板并把数据传送给它时，必须先调用 EmptyClipboard，通知 Windows 释放或删除剪贴板上的内容。不能在现有的剪贴板内容中附加其它东西，所以，从这种意义上说，剪贴板每次只能保留一个数据项。

但是，可以在 EmptyClipboard 和 CloseClipboard 调用之间多次调用 SetClipboardData，每次都使用不同的剪贴板格式。例如，如果想在剪贴板中存储一个很短的文本字符串，可以创建一个元文件设备环境，把这个文本写入元文件中；也可以创建一个大得足以容下字符串的位图，把位图选进内存设备环境中，把这个字符串写进位图中。利用这种方法可以使字符串不仅能为从剪贴板上读取文本的程序所使用，也可以为从剪贴板上读取位图和元文件的程序所使用。更进一步说，如果在写文本之前把不同的字体选进元文件设备环境或内存设备环境中，那么读取位图或元文件的程序可以使用不同字体的字符串（当然，这些程序并不能知道元文件或位图实际上包含了一个字符串）。

如果想把一些句柄写到剪贴板上，对每个句柄均可以调用 SetClipboardData:

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
SetClipboardData (CF_TEXT, hGMemText) ;
SetClipboardData (CF_BITMAP, hBitmap) ;
SetclipboardData (CF_METAFILEPICT, hGMemMFP) ;
CloseClipboard ( ) ;
```

在这三种格式的数据同时位于剪贴板上时，用 CF_TEXT 、 CF_BITMAP 或 CF_METAFILEPICT 参数调用 IsClipboardFormatAvailable 将返回 TRUE 。 通过下列调用，程序可以访问这些句柄：

```
hGMemText = GetClipboardData (CF_TEXT) ;
```

或

```
hBitmap = GetClipboardData (CF_BITMAP) ;
```

或

```
hGMemMFP-GetClipboardData (CF_METAFILEPICT) ;
```

下一次程序调用 EmptyClipboard 时，Windows 将释放或删除剪贴板上保留的所有三个句柄和作为 METAFILEPICT 结构一部分的元文件。

通过首先打开剪贴板，然后调用 EnumClipboardFormats，程序可以确定剪贴板存储的所有格式。开始时设置变量 iFormat 为：

```
iFormat = 0;
OpenClipboard (hwnd) ;
```

现在从 0 值开始逐次进行连续的 EnumClipbaordFormats 调用。函数将为当前在剪贴板中的每种格式返回一个正的 iFormat 值。当函数返回 0 时，表示完成：

```
while (iFormat = EnumClipboardFormats (iFormat))
     {
     [每个 iFormat 值的逻辑]
     }
CloseClipboard ( ) ;
```

你可以通过下面的调用来获取当前在剪贴板中的不同格式的个数：

```
iCount = CountClipboardFormats ( ) ;
```

16.2.2 延迟生成

当把数据放入剪贴板中时，一般来说要制作一份数据拷贝，并将包含这份拷贝的内存块句柄传给剪贴板。对非常大的数据项来说，这种方法会浪费内存空间。如果用户不想把数据粘贴到另一个程序里，那么，在它被其它内容取代之前，它将一直占据着内存空间。

通过使用一种叫做“延迟生成”的技术可以避免这个问题。实际上，直到另一个程序需要数据，程序才提供这份数据。为此，不将数据句柄传给 Windows，而是在 SetClipboardData 调用中使用 NULL：

```
OpenClipboard (hwnd) ;
EmptyClipboard ( ) ;
SetClipboardData (wFormat, NULL) ;
CloseClipboard ( ) ;
```

可以有多个使用不同 iFormat 值的 SetClipboardData 调用，对其中某些调用可使用 NULL 值，而对其它一些则使用实际的句柄值。

前面的过程比较简单，以下过程就要稍微复杂一些了。当另一个程序调用 GetClipboardData 时，Windows 将检查那种格式的句柄是否为 NULL 。如果是， Windows 将给“剪贴板所有者”（你的程序）发送一条消息， 请求数据的实际句柄，这时你的程序必须提供这个句柄。

更具体地说，“剪贴板所有者”是将数据放入剪贴板的最后一个窗口。当一个程序调用 OpenClipboard 时， Windows 保存调用这个函数时所用的窗口句柄，这个句柄标识打开剪贴板的窗口。一旦收到一个 EmptyClipboard 调用，Windows 就使这个窗口作为新的剪贴板所有者。

使用延迟生成技术的程序在它的窗口过程中必须处理三条消息： WM_RENDERFORMAT 、 RENDERALLFORMATS 和 WM_DESTROYCLIPBOARD 。当另一个程序调用 GetClipboardData 时，Windows 给窗口过程发送一条 WM_RENDERFORMAT 消息， wParam 的值是所要求的格式。在处理 WM_RENDERFORMAT 消息时，不要打开或清空剪贴板。为 wParam 所指定的格式创建一个全局内存块，把数据传给它，并用正确的格式和相应句柄调用 SetClipboardData 。很明显，为了在处理 WM_RENDERFORMAT 时正确地构造出此数据，需要在程序中保留这些信息。当另一个程序调用 EmptyClipboard 时，Windows 给你的程序发送一条 WM_DESTROYCLLBOARD 消息告诉你不再需要构造剪贴板数据的信息。你的程序不再是剪贴板的所有者。

如果程序在它自己仍然是剪贴板所有者的时候就要终止运行，并且剪贴板上仍然包含着该程序用 SetClipboardData 设置的 NULL 数据句柄，它将收到 WM_RENDERALLFORMATS 消息。这时，应该打开剪贴板，清空它，把数据装入内存块中，并为每种格式调用 SetClipboardata, 然后关闭剪贴板。 WM_RENDERALLFORMATS 消息是窗口过程最后收到的消息之一。它后面跟有 WM_DESTROYCLIPBOARD 消息(由于已经生成了所有数据), 然后是正常的 WM_DESTROY 消息。

如果程序只把一种格式的数据传送给剪贴板（例如文本），那么你可以把 WM_RENDERALLFORMAT 和 WM_RENDERFORMAT 处理结合在一起，这段代码如下：

```
case WM_RENDERALLFORMATS:
    OpenClipboard (hwnd) ;
    EmptyClipboard ( ) ;
                        // 进入下面的 case , 继续执行
case WM_RENDERFORMAT:
    [将文本放入全局内存块]
    SetClipboardData (CF_TEXT, hMem) ;
    if (imessage = = WM_RENDERALLFORMATS)
    CloseClipboard ( ) ;
return0:
```

如果你的程序使用几种剪贴板格式，那么你可能想为 wParam 所要求的格式处理 WM_RENDERFORMAT 。除非程序难于保存构造数据所需的信息，否则不需要处理 WM_DESTROYCLIPBOARD 消息。

16.2.3 私有数据格式

到目前为止，我们仅处理了 Windows 定义的标准剪贴板数据格式。但是，你可能想用剪贴板来存储“私有数据格式”。许多字处理程序使用这种技术来存储包含着字体和格式化信息的文本。

初看起来，这个概念似乎是没有意义的。如果剪贴板的作用是在应用程序之间传送数据，那么，为什么剪贴板中要含有只有一个应用程序才能理解的数据呢？答案很简单：剪贴板允许在同一个程序的不同实例之间传送数据，很明显，这些实例能理解它们自己的私有数据格式。

有几种使用私有数据格式的方法。最简单的方法用到一种表面上是标准剪贴板格式(文本、位图或元文件)的只对你的程序才有意义的数据。这种情况下，在 SetClipboardData 和 GetClipboardData 调用中可使用下列 iFormat 值：CF_DSPTEXT 、CF_BITMAP 或 CF_METAFILEPICT 。字母 DSP 代表“显示”——这些格式允许 CLIPBOARD 像显示文本、位图或元文件一样来显示数据。但是，另一个使用常规的 CF_TEXT 、CF_BITMAP 或 CF_METAFLLEPICT 格式调用 GetClipboardData 的程序不能获取这个数据。

如果用其中一种格式把数据放入剪贴板中，则必须使用同样的格式读出数据。但是，如何知道数据是来自程序的另一个实例还是来自使用其中某种数据格式的另一个程序呢？这里有一种方法，可以通过下列调用首先获得剪贴板所有者：

```
hwndClipOwner = GetClipboardOwner ( ) ;
```

然后可以得到此窗口句柄的窗口类名称:

```
char szClassName[16] ;
[其它程序行]
GetClassName (hwndClipOwner, &szClassName, 16) ;
```

如果类名与程序名相同，那么数据是由程序的另一个实例传送到剪贴板中的。

使用私有数据格式的第二种方法涉及到 CF_OWNERDISPLAY 标志。SetClipboardData 的内存句柄是 NULL:

```
SetClipboardData (CF_OWNERDISPLAY, NULL) ;
```

这是某些字处理器在 Windows 95 的剪贴板浏览器的客户区域中显示格式化文本时所采用的方法。很明显，剪贴板浏览器不知道如何显示这种格式化文本。当一个字处理器指定 CF_OWNERDISPLAY 格式时，它也就承担起在剪贴板浏览器的客户区域中绘图的责任。

由于内存句柄为 NULL，所以用 CF_OWNERDISPLAY 格式调用 SetClipboardData 的程序必须处理由 Windows 发往剪贴板所有者的延迟生成消息以及 5 条附加消息。这 5 条消息是由剪贴板浏览器发送到剪贴板所有者的：

- WM_ASKCBFORMATNAME——剪贴板浏览器把这条消息发送到剪贴板所有者，以得到数据格式名称。lParam 参数是指向缓冲区的指针，wParam 是这个缓冲区

能容纳的最大字符数目。剪贴板所有者必须把剪贴板数据格式的名字拷贝到这个缓冲区中。

- WM_SIZECLIPBOARD——这条消息通知剪贴板所有者，剪贴板浏览器的客户区域的大小已发生了变化。wParam 参数是剪贴板浏览器的句柄，lParam 是指向包含新尺寸的 RECT 结构的指针。如果 RECT 结构中都是 0，则剪贴板浏览器退出或变成了一个图标。尽管 Windows 的剪贴板浏览器只允许它自己的一个实例运行，但其它剪贴板浏览器也能把这条消息发送给剪贴板所有者。对付多个剪贴板浏览器并非不可能(假定 wParam 标识特定的浏览器)，但剪贴板所有者处理起来也不容易。
- WM_PAINTCLIPBOARD——这条消息通知剪贴板所有者修改剪贴板浏览器的客户区。同时，wParam 是剪贴板浏览器窗口的句柄，lParam 是指向 PAINTSTRUCT 结构的指针。剪贴板所有者可以从这种结构的 hdc 域中得到剪贴板浏览器设备环境句柄。
- WM_HSCROLLCLIPBOARD 和 WM_VSCROLLCLIPBOARD——这两条消息通知剪贴板所有者，用户已经滚动了剪贴板浏览器的滚动条。wParam 参数是剪贴板浏览器窗口的句柄，lParam 的低位字是滚动请求，并且，如果低位字是 SB_THUMBPOSITION，那么 lParam 的高位字是滑块位置。

处理这些消息比较麻烦，看来并不值得这样做。但是，这种处理对用户来说是有益的：当从字处理器把文本拷贝到剪贴板时，用户在剪贴板浏览器的客户区中看见文本还保持着格式心里会舒坦些。

使用私有剪贴板数据格式的第三种方法是注册自己的剪贴板格式名。你提供格式名给 Windows，Windows 给程序提供一个序号，它可以用作 SetClipboardData 和 GetClipboardData 的格式参数。一般来说，采用这种方法的程序也要以一种标准格式把数据拷贝到剪贴板中。这种方法允许剪贴板浏览器在它的客户区中显示数据，并且允许其它程序从剪贴板上拷贝数据。

例如，假定我们已经编写了一个以位图格式、元文件格式和自己的已注册的剪贴板格式把数据拷贝到剪贴板中去的矢量绘图程序。剪贴板浏览器将显示元文件，其它从剪贴板上读取位图和元文件的程序将获得这几种格式。但是，当我们的矢量绘图程序需要从剪贴板上读数据时，它会按照自己已注册的格式拷贝数据，这是因为这种格式可能包含着比位图文件或者元文件更多的信息。

程序通过下面调用注册一个新的剪贴板格式：

```
iFormat = RegisterClipboardForamat (pszFormatName) ;
```

iFormat 的值介于 0xC000 和 0xFFFF 之间。剪贴板浏览器(或一个通过调用 EnumClipboardFormats 获取当前所有剪贴板数据格式的程序)可以获取这种数据格式的 ASCII 名，这是通过下面调用实现的：

```
GetClipboardFormatName (iFormat, psBuffer, iMaxCount) ;
```

Windows 将多达 iMaxCount 个字符拷贝到 psBuffer 中。

使用这种方法把数据拷贝到剪贴板中的程序员可能需要公开数据格式名称和实际的数据格式。如果这个程序流行起来，那么其它程序就会以这种格式从剪贴板拷贝数据。

16.3　实现剪贴板浏览器

监视剪贴板内容变化的程序称为"剪贴板浏览器"。你可以在 Windows 95 中得到一个剪贴板浏览器，但是你也可以编写自己的剪贴板浏览器程序。剪贴板浏览器通过传递到浏览器窗口过程的消息来监视剪贴板内容的变化。

16.3.1　剪贴板浏览器链

任意数量的剪贴板浏览器应用程序都可以同时在 Windows 下运行，它们都可以监视剪贴板内容的变化。但是，从 Windows 的角度来看，只存在一个剪贴板浏览器，我们称之为"当前剪辑板浏览器"。Windows 只保留一个识别当前剪贴板浏览器的窗口句柄，并且当剪贴板的内容发生变化时只把消息发送到那个窗口中。

剪贴板浏览器应用程序有必要加入"剪贴板浏览链"，以便运行的所有剪贴板浏览器都可以收到 Windows 发送给当前剪贴板浏览器的消息。当一个程序注册自己为一个剪贴板浏览器时，它就成为当前剪贴板浏览器。Windows 把先前的当前浏览器的窗口句柄交给这个程序，并且此程序保存这个句柄。当此程序收到一条剪贴板浏鉴器消息时，它把这条消息发送给剪贴板链中下一个程序的窗口过程。

16.3.2　剪贴板浏览器的函数和消息

程序通过调用 SetClipboardViewer 函数可以成为剪贴板浏览器链的一部分。如果程序的主要作用是作为剪贴板浏览器，那么这个程序在处理 WM_CREATE 消息期间可以调用这个函数，该函数返回前一个当前剪贴板浏览器的窗口句柄。程序应该把这个句柄保存在静态变量中：

```
static HWND hwndNextViewer;
[其它程序行]
case WM_CREATE:
     [其它程序行]

     hwndNextViewer = SetClipboardViewer (hwnd) ;
```

如果在 Windows 的一次运行期间你的程序是成为剪贴板浏览器的第一个程序，那么 hwndNextViewer 将为 NULL 。

不管剪贴板中的内容怎样变化，Windows 都将把 WM_DRAWCLIPBOARD 消息发送给当前剪贴板浏览器(最近注册为剪贴板浏览器的窗口)。剪贴板浏览器链中的每个程序都

应该用 SendMessage 把这条消息发送到下一个剪贴板浏览器。浏览器链中的最后一个程序（第一个将自己注册为剪贴板浏览器的窗口）所保存的 hwndNextViewer 为 NULL 。如果 hwndNextViewer 为 NULL, 那么程序只简单地返回, 而不发送任何消息给其它程序(不要把 WM_DRAWCLLPBOARD 消息和 WM_PAINTCLIPBOARD 消息混淆起来。WM_PAINTCLIPBOARD 是由剪贴板浏览器发送给使用 CF_OWNERDISPLAY 剪贴板数据格式的程序的，而 WM_DRAWCLIPBOARD 消息是由 Windows 发往当前剪贴板浏览器的）。

处理 WM_DRAWCLIPBOARD 消息的最简单的方法是将消息发送给下一个剪贴板浏览器（除非 hwndNextViewer 为 NULL ）并使窗口的客户区域无效:

```
case WM_DRAWCLIPBOARD
     if (hwndNextViewer)
          SendMessage (hwndNextViewer, iMsg, wParam, lParam) ;

     InvalidateRect (hwnd, NULL, TRUE)
     return 0;
```

在处理 WM_PAINT 消息期间, 通过使用常规的 OpenClipboard 、 GetClipboardData 和 CloseClipboard 调用可以读取剪贴板的内容。

当某个程序想从剪贴板浏览器链中删除它自己时，它必须调用 ChangeClipboardChain 。这个函数接收脱离浏览器链的程序的窗口句柄和下一个剪贴板浏览器的窗口句柄：

```
ChangeClipboardChair (hwndNextViewer) ;
```

当程序调用 ChangeClipboardChain 时, Windows 发送 WM_CHANGECLIPBOARD 消息给当前剪贴板浏览器。 wParam 参数是从链中移去它自己的那个浏览器的窗口句柄 (ChangeClipboardChain 的第一个参数), lParam 是从链中移去自己后的下一个剪贴板浏览器的窗口句柄 (ChangeClipboardChainChain 的第二个参数)。

当程序接收到 WM_CHANGECLIPBOARD 消息时, 必须检查 wParam 是否等于已经保存的 hwndNextViewer 的值。如果是这样，程序必须设置 hwndNextViewer 为 lParam 。这项工作保证将来的 WM_DRAWCLIPBOARD 消息不会发送给从剪贴板浏览器链中删除了自己的窗口。如果 wParam 不等于 hwndNextViewer, 并且 hwndNextViewer 不为 NULL, 则把消息送到下一个剪贴板浏览器。

```
case WM_CHANGECBCHAIN,
     if (wParam = = hwndNextViewer)
          hwndNextViewer = LOWORD (lParam) ;

     else if (hwndNextViewer)
          SendMessage (hwndNextViewer, iMsg, wParam, lparam) ;
raeturn 0;
```

不一定要使用 else if 语句, 它只用于保证 hwndNextViewer 为非 NULL 的值。

hwndNextViewer 的值为 NULL 时, 执行这段代码的程序就是链中最后一个浏览器, 而

这是不可能的。

当程序快结束时，如果它仍然在剪贴板浏览器链中，则必须从链中删除它。你可以在处理 WM_DESTROY 消息时调用 ChangeClipboardChain 来完成这项工作。

```
case WM_DESTROY:
    ChangeClipboardChain (hwnd, hwndNextViever) ;
    PostQuitMessage (0) ;
retun 0;
```

Windows 还有一个允许程序获得第一个剪贴板浏览器窗口句柄的函数:

```
hwndViewer = GetClipboardViewer ( ) ;
```

一般来说不需要这个函数。如果没有当前剪贴板浏览器，则返回值为 NULL 。

下面是一个说明剪贴板浏览器链如何工作的例子。当 Windows 刚启动时, 当前剪贴板浏览器是 NULL:

当前编辑板浏览器: NULL

一个具有 hwnd1 窗口句柄的程序调用 SetClipboardViewer 。这个函数返回的 NULL 成为这个程序中的 hwndNextViewer 值：

当前剪贴板浏览器: *hwnd1*
hwnd1 的下一个浏览器: NULL

另一个具有 hwnd2 的程序调用 SetClipboardViewer 。这个函数返回 hwnd1 ：

当前剪贴板浏览器: *hwnd2*
hwnd2 的下一个浏览器: *hwnd1*
hwnd1 的下一个浏览器: NULL

第三个程序（hwnd3)和第四个程序（hwnd4)也调用 SetClipboardViewer, 并且返回 hwnd2 和 hwnd3:

当前剪贴板浏浏器: *hwnd4*
hwnd4 的下一个浏览器: *hwnd3*
hwnd3 的下一个浏览器: *hwnd2*
hwnd2 的下一个浏览器: *hwnd1*
hwnd1 的下一个浏览器: NULL

当剪贴板的内容发生变化时, Windows 发送一条 WM_DRAWCLIPBOARD 消息给 hwnd4 , hwnd4 发送消息给 hwnd3 ， hwnd3 发送消息给 hwnd2 ， hwnd2 发送消息给 hwnd1 ， hwnd1 返回。

现在, hwnd2 决定通过下列调用从链中删除自己:

```
ChangClipboardChain (hwnd2, hwnd1) ;
```

Windows 将 wParam 等于 hwnd2、 lParam 等于 hwnd1 的 WM_CHANGECBCHAIN 消息发送给 hwnd4。由于 hwnd4 的下一个浏览器是 hwnd3，所以 hwnd4 把这条消息传给 hwnd3。现在 hwnd3 注意到 wParam 等于它的下一个浏览器(hwnd2)，所以将下一个浏览器设置为 lParam (hwnd1)并且返回。这样工作就完成了。现在剪贴板浏览器链如下：

当前剪贴板浏览器:	*hwnd4*
hwnd4 的下一个浏览器:	*hwnd3*
hwnd3 的下一个浏览器:	*hwnd1*
hwnd1 的下一个浏览器:	NULL

16. 3. 3 一个简单的剪贴板浏览器

剪贴板浏览器不一定要像 Windows 95 所提供的那样完善，例如，剪贴板浏览器可以只显示一种剪贴板数据格式。图 16-1 中所示的 CLIPVIEW 程序是一种只能显示 CF_TEXT 格式的剪贴板浏览器。

CLIPVIEW.MAK

```
#------------------------
# CLIPVIEW.MAK make file
#------------------------

clipview.exe : clipview.obj
     $(LINKER) $(GUIFLAGS) -OUT:clipview.exe clipview.obj $(GUILIBS)

clipview.obj : clipview.c
     $(CC) $(CFLAGS) clipview.c
```

CLIPVIEW.C

```
/*-----------------------------------------
   CLIPVIEW.C -- Simple Clipboard Viewer
                 (c) Charles Petzold, 1996
  -----------------------------------------*/
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "ClipView" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
     wndclass.cbWndExtra     = 0 ;
     wndclass.hInstance      = hInstance ;
     wndclass.hIcon          = NULL ;
     wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName   = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Simple Clipboard Viewer (Text Only)",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HWND hwndNextViewer ;
     HGLOBAL     hGMem ;
     HDC         hdc ;
     PSTR        pGMem ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_CREATE :
               hwndNextViewer = SetClipboardViewer (hwnd) ;
               return 0 ;

          case WM_CHANGECBCHAIN :
               if ((HWND) wParam == hwndNextViewer)
                    hwndNextViewer = (HWND) lParam ;

               else if (hwndNextViewer)
                    SendMessage (hwndNextViewer, iMsg, wParam, lParam) ;

               return 0 ;

          case WM_DRAWCLIPBOARD :
               if (hwndNextViewer)
                    SendMessage (hwndNextViewer, iMsg, wParam, lParam) ;
```

```
                InvalidateRect (hwnd, NULL, TRUE) ;
                return 0 ;

           case WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;
                GetClientRect (hwnd, &rect) ;
                OpenClipboard (hwnd) ;

                hGMem = GetClipboardData (CF_TEXT) ;

                if (hGMem != NULL)
                     {
                     pGMem = (PSTR) GlobalLock (hGMem) ;
                     DrawText (hdc, pGMem, -1, &rect, DT_EXPANDTABS) ;
                     GlobalUnlock (hGMem) ;
                     }

                CloseClipboard () ;
                EndPaint (hwnd, &ps) ;
                return 0 ;

           case WM_DESTROY :
                ChangeClipboardChain (hwnd, hwndNextViewer) ;
                PostQuitMessage (0) ;
                return 0 ;
           }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
      }
```

图 16-1 CLIPVIEW程序

CLIPVIEW 按上面所讨论的方法来处理 WM_CREATE、WM_CHANGECBCHALN、WM_DRAWCLIPBOARD 和 WM_DESTROY 消息。WM_PAINT 消息处理打开剪贴板，并用 CF_TEXT 格式调用 GetClipboardData 。如果函数返回一个内存句柄，那么 CLIPVIEW 将锁定它，并用 DrawText 在客户区显示文本。

处理五种标准格式以外的数据格式的剪贴板浏览器还需要完成其它一些工作（如 Windows 95 提供的那个剪贴板一样），比如显示剪贴板中当前所有数据格式的名称。用户可以通过调用 EnumClipboardFormats 并使用 GetClipboardFormatNome 得到非标准数据格式名称来完成这项工作。使用 CF_OWNERDISPLAY 数据格式的剪贴板浏览器必须把下面四条消息送往剪贴板数据的拥有者以显示该数据:

```
WM_PAINTCLIPBOARD    WM_VSCROLLCLLPBOARD
WM_SIZECLIPBOARD      WM_HSCROLLCLIPBOARD
```

如果你想编写这样的剪贴板浏览器，那么必须使用 GetClipboardOwner 获得剪贴板所有者的窗口句柄，并当你需要修改剪贴板的客户区时，必须将这些消息发送给该窗口。

第 17 章　动态数据交换(DDE)

动态数据交换(DDE)是 Windows 95 支持的几种进程间通讯机制之一，其它三种机制是 Windows 剪贴板(在第 16 章讨论)，动态链接库中的共享内存(在第 19 章讨论)和 OLE （在第 20 章讨论）。DDE 的功能没有 OLE 强大，不过，一般要比 OLE 容易实现。

DDE 基于 Windows 的消息系统。两个 Windows 程序通过相互之间传递消息进行“对话”，这两个程序被称为“服务器”和“客户”。DDE 服务器是一个维护着其它 Windows 程序可能使用的数据的程序，DDE 客户是从服务器获得这些数据的程序。

在 Windows95 中，程序可以选择性地使用在某些方面简化了的 DDE 管理库(DDEML)。DDEML 通过提供高级函数调用层，从而把程序与 DDE 的消息机制隔离开来。使用 DDEML 的程序使用回调函数响应 DDE 消息。由于 DDEML 创建在 DDE 消息传递系统上，所以两种使用 DDE 的方法是兼容的。

本章将首先讨论使用 DDE 的传统方法，然后说明如何用 DDEML 解决进程间的通讯问题。

DDE 对话是由客户程序发动的。客户程序把一条消息（称为 WM_DDE_INITIATE ）发给当前运行的所有 Windows 程序，这条消息指明了客户程序所需要的数据类别，拥有这些数据的 DDE 服务器可以响应这条消息。这样，对话就开始了。

同一个 Windows 程序既可以是一个程序的客户,也可以是另一个程序的服务器，但这需要两种不同的 DDE 对话。一个服务器可以把数据传给多个客户，一个客户可以从多个服务器获取数据，同样，这也需要多种 DDE 对话。为了确保这种对话是唯一的和互不影响的，每一对话（在客户和服务器两边）都使用不同的窗口。一般来说，支持 DDE 的程序将为其所维护的每种对话创建一个隐藏的子窗口。

DDE 对话涉及到的程序不需要为了协同工作而进行特殊编程。正如下一节所要讨论的那样，一般来说，DDE 服务器的作者会公开数据是如何被识别的。DDE 客户程序的用户(例如 Microsoft Excel)可以利用这种信息建立两个程序之间的 DDE 对话。

如果要编写两个或多个必须相互通讯而不与其它 Windows 程序进行通讯的 Windows 程序，就可考虑定义自己的通讯协议。但是，如果两个或多个程序必须既要读共享数据又要写入它，那么你必须将数据放在一个内存映象文件中。我们将在本章后面看到，DDE 客户从来不写数据，而只是读数据。因此一个 DDE 服务器可以安全地使用 GlobalAlloc 或者 HeapAlloc 函数分配内存。

由于 DDE 采用了 Windows 自身的消息系统，所以它显得非常自然。但这并不表明 DDE 很容易实现，协议的设计涉及很多因素，程序必须准备好处理一些相当棘手的问题。

17.1 基本概念

当客户向服务器请求数据时,它必须能标识它所希望的数据类型，用分别称为“应用程序”、数据“主题”和数据“项”的三个字符串可以完成这项工作。

17.1.1 应用程序、主题和项

用一个例子可以很好地说明应用程序、主题和项的概念。在这一章的前半部分将讨论如何编写一个称为 DDEPOP1 的 Windows DDE 服务器程序,这个程序包含 1970、1980、1990 年美国人口普查所得到的人口数据。根据平方外插法，程序可以立即计算出任何一个州或整个美国（“此刻”）的人口数。

任何一个编写 DDE 服务器程序的程序员都应该说明如何用这三个字符串来标识数据:

- 服务器应用程序名: 在这个示例中是“ DDEPOP1 ”。每个服务器程序只有一个应用程序名。
- 主题名:所有的 DDE 服务器至少支持一个主题。在 DDEPOP1 中仅支持一个主题，它是通过字符串“ US_Poplation ”来标识的。可以想象，DDEPOP1 可以扩充到包含与某个州内每平方公里面积有关的数据，在这种情况下，程序支持叫做“ US_Area ”的第二个主题。
- 项名：在每个主题中，DDE 服务器支持一个或多个数据项。在 DDEPOP1 中，数据项用标准的双字符邮政缩写法来标识州名，例如，“ NY ”为纽约，“ CA ”为加利福尼亚，“ US ”为美国。DDEPOP1 支持 52 项——50 个州、哥伦比亚特区（“ DC ”）和全美国。

这些说明对使用 DDEPOP1 服务器的另一个可以作为客户的 Windows 程序(如 Microsoft Excel)来说足够了。为了将 DDEPOP1 用于 Microsoft Excel，只需在电子表格中填入:

```
=DDEPOP1|US_Population!US
```

这三个字符串指明了应用程序、主题和数据项（在本程序中指全美人口数）。如果 DDEPOP1.EXE 不在运行，那么 Microseft Excel 将试图启动它（DDEPOP1 必须在当前目录中或在 PATH 环境变量所列的目录中)。如果成功，EXCEL 将发动与 DDEPOP1 的对话，获取人口数据，在表格中显示人口数据。这些人口数据可以被格式化、图形化，也可用于计算目的。

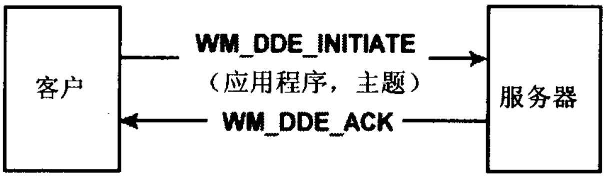
最令人感兴趣的是可以在电子表格中周期性地更新人口数据,这就是人们所熟知的“热链接”或者（一种略微不同的）“温链接”。每隔 5 秒钟，DDEPOP1 重新计算数据，并在数据项发生变化时通知客户程序。在计算全美人口时，可以看到大约每隔 15 秒钟就有 1 人诞生。

17.1.2 对话的类型

DDE 对话的基本类型有三种：冷链接、热链接和温链接。这些对话使用 DDE.H 头文件中定义的 DDE 消息。最简单的对话类型是冷链接。

1. 冷链接

当客户广播一条标明它所需要的应用程序和主题的 WM_INITIATE 消息时，冷链接对话就开始了(应用程序和主题均可置为 NULL，以便与任何服务器应用程序或者根据任何主题进行对话)。支持特定主题的服务器应用程序用一条 WM_DDE_ACK（“确认”）消息来响应客户程序。



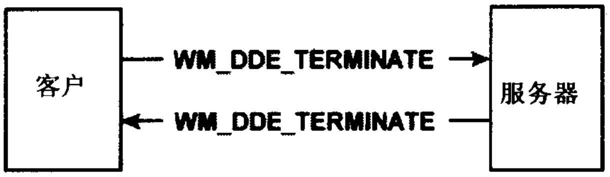
接着，客户程序通过邮递（post）一条 WM_DDE_REQUEST 消息来请求特定的数据项。如果服务器能提供这个数据项，它就通过邮递 WM_DDE_DATA 消息来响应客户：

上图指明，客户可以回答服务器说它已经接收到了 WM_DDE_DATA 消息，这是可选的（因此在上图中，我们把 WM_DDE_ACK 消息放在括号中）。服务器用 WM_DDE_DATA 消息中的一个标志表明它是否希望此确认。随 WM_DDE_ACK 消息传递的标志指示一个“肯定的”回答。

如果客户把一条 WM_DDE_REQUEST 消息邮递给服务器, 并且服务器不能提供所需要的数据项, 那么服务器邮递一条“否定的”的 WM_DDE_ACK 消息给客户:

当客户程序把 WM_DDE_REQUEST 消息邮递给服务器(对同一数据项或不同数据

项)时，DDE 对话继续进行，服务器用 WM_DDE_DATA 或 WM_DDE_ACK 消息作出响应。当客户和服务器相互之间邮递 WM_DDE_TERMINATE 消息时，对话结束:

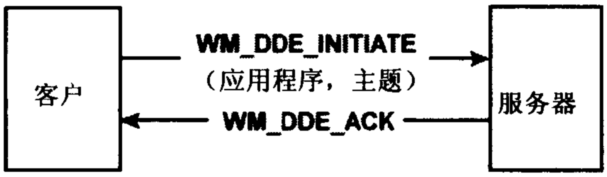


尽管上图中是客户首先邮递 WM_DDE_TERMINATE 消息，但情况并非总是如此。服务器可以首先邮递 WM_DDE_TERMINATE 消息,并且客户必须对它进行响应。

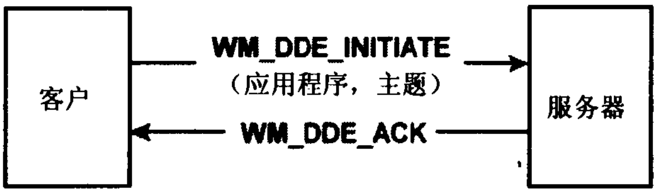
2.　热链接

冷链接的问题之一是服务器程序已经访问的数据可能会随着时间的推移发生变化(在 DDEPOP1 中就是这样)。在冷链接中，客户程序不知道数据何时发生变化。热链接能够解决这个问题。

同样，DDE 对话从 WM_DDE_INITLATE 和 WM_DDE_ACK 消息开始:



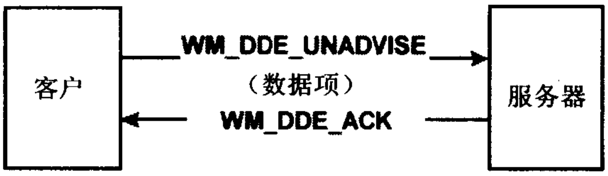
客户通过把消息 WM_DDE_ADVISE 邮递给服务器来说明它所需要的数据项。如果服务器可以得到这个数据项，那么它通过邮递一条 WM_DDE_ACK 消息作出响应：



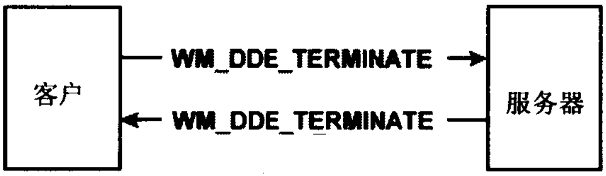
肯定确认说明服务器能够提供数据；否定确认说明它不能提供数据。

至此，当数据项发生变化时，服务器有责任通知客户程序。此通知是利用 WM_DDE_DATA 消息进行的，客户可能会也可能不会用 WM_DDE_ACK 消息进行响应:

当客户程序不再希望得到数据项被修改的通知时，它就发送一条WM_DDE_UNADVISE 消息给服务器，服务器将作出确认:



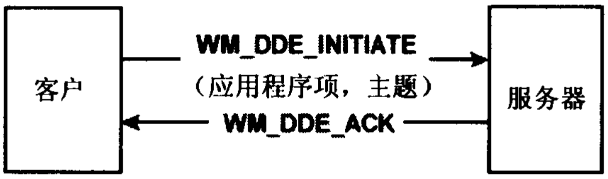
当邮递 WM_DDE_TERMINATE 消息时,对话终止。



冷链接和热链接不是互斥的。在同一个 DDE 对话期间，客户可以通过使用WM_DDE_REQUEST 消息(用于冷链接)来请求一些数据项,也可以通过使用WM_DDE_ADVISE 消息(用于热链接)来请求另一些数据项。

3. 温链接

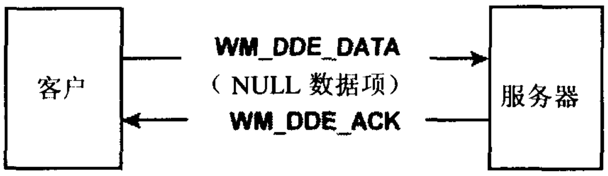
温链接结合了冷链接和热链接的成份。对话正常开始:



和热链接一样，客户邮递 WM_DDE_ADVISE 消息给服务器，服务器作出肯定或否定确认:

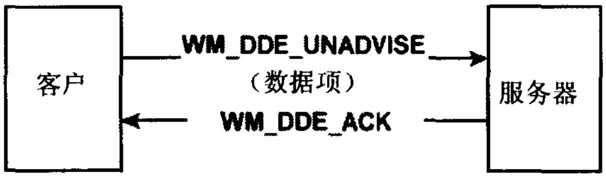
但是，通过 WM_DDE_ADVISE 消息传递的标志说明客户只希望被通知数据发生

了变化而不是立刻接收新的数据项，所以服务器邮递带有 NULL 数据的 WM_DDE_DATA 消息:

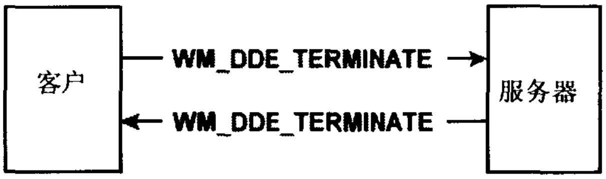


现在客户知道特定的数据项已经发生变化。为了获取这个数据项，与冷链接一样，客户使用 WM_DDE_REQUEST 消息:

与在热链接中一样，客户通过邮递 WM_DDE_UNADVISE 消息给服务器，从而终止数据项发生变化的通知:



用 WM_DDE_TERMINATE 消息终止对话:



这三种类型的对话使用了除下面两条消息之外的所有 DDE 消息: WM_DDE_POKE(客户向服务器发送未经请求的数据)和 WM_DDE_EXECUTE(客户发送命令串给服务器)。本章不讨论这两条消息。

DDE.H 头文件还定义了四个结构:

- DDEACK(用于 WM_DDE_ACK 消息中)
- DDEADVISE(用于 WM_DDE_ADVISE 消息中)
- DDEPOKE(用于 WM_DDE_DATA 消息中)
- DDEPOKE(用于 WM_DDE_POKE 消息中)

在本章讨论的示例程序中用到前三种结构。

17.1.3　字符串和原子

前面已讨论了 DDE 客户和服务器如何用三个字符串——应用程序、主题和数据项——来标识数据，但在客户和服务器之间的实际消息中，这些字符串并不出现，而是使用“原子”代替它们。

在不区分大小写的情况下，原子是用来指定字符串的 WORD 类型的值。在用户程序中可以用原子来代替字符串，在这种情况下，原子表（用字符串引用原子值的表）被存储在程序的数据空间中。

可以如下定义一个原子：

```
ATOM aAtom ;
```

用下列函数可以把一个字符串加到原子表中:

```
aAtom = AddAtom (lpString) ;
```

如果字符串没有包含在原子表中，那么此函数把这个字符串加入到原子表中，且返回标识此字符串的唯一值。每个原子都有一个“引用计数”，它是用相同字符串调用 AddAtom 的次数。引用计数的初值被置为 1 。如果字符串已经存在于原子表中，则函数返回标识字符串的数值，且递增引用计数。

函数：

```
DeleteAtom (aAtom) ;
```

递减引用计数。当计数为 0 时，从原子表中删去原子和字符串。

函数：

```
aAtom = FindAtom (lpString) ;
```

将返回与字符串相关联的原子(如果字符串不在原子表中，则返回 0)。这个函数不影响原子的引用计数。

函数：

```
iBytes = GetAtomName (aAtom, pBuffer, iBufferSize) ;
```

返回与原子相关联的字符串。最后一个参数是第二个参数指定的缓冲区的大小。函数返回拷贝到缓冲区中的字节数，它不影响引用计数。

这四个函数（还有一些不太重要的函数）允许在程序中使用原子。但是，由于原子表存放在程序的数据空间中，因此各个原子在你的程序中是唯一的。为了在 DDE 中使用原子，必须使用与上面所述的函数相似的四个函数:

```
aAtom = GlobalAddAtom (pString) ;
```

```
GlobalDeleteAtom (aAtom) ;
aAtom = GlobalFindAtom (pSTRING) ;
iBytes = GlobalGetAtomName (aAtom, pBuffer, iBufferSize) ;
```

这些原子的原子表存放在 Windows 动态链接库的共享数据段内，因此对于所有 Windows 程序来说是公共的，一个程序可以用 GlobalAddAtom 在原子表中增加一个字符，并把原子传送到另一个程序中，后者又可以用 GlobalGetAtomName 获得与原子相关联的字符串。这就是 Windows 的程序识别 DDE 应用程序、主题和数据项的方法。

DDE 中使用原子的规则是非常重要的：如果当某个程序仍然需要某个原子，另一个程序却从原子表中把它删除掉了，这显然是不合适的；如果不再需要某个原子，却没有从原子表中删除它，这显然也是不合适的。由于这些原因，在程序中必须小心处理原子。

原子可用于 DDE 应用程序、主题和数据项字符串。从一个 Windows 程序传送到另一个 Windows 程序的数据结构可以使用 GlobalAlloc 或者 HeapAlloc 从堆上分配，也可以作为内存映象文件。这两种方法都允许数据被多个 Windows 程序共享。然而，我不推荐为 DDE 对话使用内存映象文件，因为它们给客户提供了覆盖或者破坏数据的机会。通过从堆上分配一个内存块，服务器保证了数据对所有的客户都是“只读”的。控制哪个程序负责分配和释放这些内存块的 DDE 规则也是相当严格的。

在 Win32 中， GlobalAlloc 只是 HeapAlloc 的一种包装，而且它保留的内存对调用者来说是私有的，其它进程不能接触到它。即使 GMEM_DDESHARE （和 GMEM_SHARE ）标志也不会造成任何区别，因为 GlobalAlloc 忽略这些标志。那么一个 DDE 客户如何才能看到存储在服务器私有内存中的数据呢？它不能。当服务器给客户发送一条 WM_DDE_DATA 消息时， Windows 对所引用的包含数据的 DDEDATA 结构作一份拷贝。客户看到的是所分配内存块的一份拷贝，而不是内存块本身。

17.2 一个 DDE 服务器程序

DDEPOP1 程序是一个能够为 DDE 客户程序提供瞬时州人口数量的 DDE 服务器，这个程序如图 17-1 所示。注意，虽然 GlobalAlloc 不要求 GMEM_DDESHARE 标志，但是为了清晰起见，在下面的源代码中仍然包含了这个标志。

DDEPOP1.MAK

```
#-----------------------
# DDEPOP1.MAK make file
#-----------------------

ddepop1.exe : ddepop1.obj ddepop1.res
     $(LINKER) $(GUIFLAGS) -OUT:ddepop1.exe ddepop1.obj ddepop1.res $(GUILIBS)

ddepop1.obj : ddepop1.c ddepop.h
     $(CC) $(CFLAGS) ddepop1.c

ddepop1.res : ddepop1.rc ddepop.ico
```

```
    $(RC) $(RCVARS) ddepop1.rc
```

DDEPOP.C

```
/*---------------------------------------------
   DDEPOP1.C -- DDE Server for Population Data
                (c) Charles Petzold, 1996
  ---------------------------------------------*/

#include <windows.h>
#include <dde.h>
#include <string.h>
#include "ddepop.h"

typedef struct
     {
     unsigned int fAdvise:1 ;
     unsigned int fDeferUpd:1 ;
     unsigned int fAckReq:1 ;
     unsigned int dummy:13 ;
     long         lPopPrev ;
     }
     POPADVISE ;

#define ID_TIMER    1
#define DDE_TIMEOUT 3000

LRESULT CALLBACK WndProc         (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK ServerProc      (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK TimerEnumProc   (HWND, LONG) ;
BOOL    CALLBACK CloseEnumProc   (HWND, LONG) ;
BOOL             PostDataMessage (HWND, HWND, int, BOOL, BOOL, BOOL) ;

char      szAppName[]     = "DdePop1" ;
char      szServerClass[] = "DdePop1.Server" ;
HINSTANCE hInst ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     hInst = hInstance ;

               // Register window class

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = 0 ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
```

```
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

               // Register window class for DDE Server

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = 0 ;
     wndclass.lpfnWndProc   = ServerProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 2 * sizeof (DWORD) ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = NULL ;
     wndclass.hCursor       = NULL ;
     wndclass.hbrBackground = NULL ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szServerClass ;
     wndclass.hIconSm       = NULL ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "DDE Population Server",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     InitPops () ;                    // initialize 'pop' structure

     SetTimer (hwnd, ID_TIMER, 5000, NULL) ;

     ShowWindow (hwnd, SW_SHOWMINNOACTIVE) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     KillTimer (hwnd, ID_TIMER) ;

     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char szTopic[] = "US_Population" ;
     ATOM        aApp, aTop ;
     HWND        hwndClient, hwndServer ;
```

```
switch (iMsg)
     {
     case WM_DDE_INITIATE :

               // wParam          -- sending window handle
               // LOWORD (lParam) -- application atom
               // HIWORD (lParam) -- topic atom

          hwndClient = (HWND) wParam ;

          aApp = GlobalAddAtom (szAppName) ;
          aTop = GlobalAddAtom (szTopic) ;

               // Check for matching atoms, create window, and
               //   acknowledge

          if ((LOWORD (lParam) == NULL || LOWORD (lParam) == aApp) &&
              (HIWORD (lParam) == NULL || HIWORD (lParam) == aTop))
               {
               hwndServer = CreateWindow (szServerClass, NULL,
                                          WS_CHILD, 0, 0, 0, 0,
                                          hwnd, NULL, hInst, NULL) ;

               SetWindowLong (hwndServer, 0, (LONG) hwndClient) ;
               SendMessage ((HWND) wParam, WM_DDE_ACK,
                            (WPARAM) hwndServer,
                            MAKELPARAM (aApp, aTop)) ;
               }

               // Otherwise, delete the atoms just created

          else
               {
               GlobalDeleteAtom (aApp) ;
               GlobalDeleteAtom (aTop) ;
               }

          return 0 ;

     case WM_TIMER :
     case WM_TIMECHANGE :

               // Calculate new current populations

          CalcPops () ;

               // Notify all child windows

          EnumChildWindows (hwnd, &TimerEnumProc, 0L) ;
          return 0 ;

     case WM_QUERYOPEN :
          return 0 ;

     case WM_CLOSE :
```

```
                    // Notify all child windows

               EnumChildWindows (hwnd, &CloseEnumProc, 0L) ;

               break ;                    // for default processing

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

LRESULT CALLBACK ServerProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     ATOM          aItem ;
     char          szItem[10] ;
     DDEACK        DdeAck ;
     DDEADVISE    *pDdeAdvise ;
     DWORD         dwTime ;
     GLOBALHANDLE  hPopAdvise, hDdeAdvise, hCommands, hDdePoke ;
     int           i ;
     UINT          uiLow, uiHi ;
     HWND          hwndClient ;
     MSG           msg ;
     POPADVISE    *pPopAdvise ;
     WORD          cfFormat, wStatus ;

     switch (iMsg)
          {
          case WM_CREATE :

                    // Allocate memory for POPADVISE structures

               hPopAdvise = GlobalAlloc (GHND, NUM_STATES * sizeof (POPADVISE)) ;

               if (hPopAdvise == NULL)
                    DestroyWindow (hwnd) ;
               else
                    SetWindowLong (hwnd, 4, (LONG) hPopAdvise) ;

               return 0 ;

          case WM_DDE_REQUEST :

                    // wParam          -- sending window handle
                    // LOWORD (lParam) -- data format
                    // HIWORD (lParam) -- item atom

               hwndClient = (HWND) wParam ;
               cfFormat   = LOWORD (lParam) ;
               aItem      = HIWORD (lParam) ;

                    // Check for matching format and data item
```

```
          if (cfFormat == CF_TEXT)
               {
               GlobalGetAtomName (aItem, szItem, sizeof (szItem)) ;

               for (i = 0 ; i < NUM_STATES ; i++)
                    if (strcmp (szItem, pop[i].szState) == 0)
                         break ;

               if (i < NUM_STATES)
                    {
                    GlobalDeleteAtom (aItem) ;
                    PostDataMessage (hwnd, hwndClient, i,
                                     FALSE, FALSE, TRUE) ;
                    return 0 ;
                    }
               }

               // Negative acknowledge if no match

          DdeAck.bAppReturnCode = 0 ;
          DdeAck.reserved       = 0 ;
          DdeAck.fBusy          = FALSE ;
          DdeAck.fAck           = FALSE ;

          wStatus = *((WORD *) &DdeAck) ;

          if (!PostMessage (hwndClient, WM_DDE_ACK, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ACK, wStatus, aItem)))
               {
               GlobalDeleteAtom (aItem) ;
               }

          return 0 ;

     case WM_DDE_ADVISE :

               // wParam -- sending window handle
               // lParam -- DDEADVISE memory handle & item atom

          UnpackDDElParam (WM_DDE_ADVISE, lParam, &uiLow, &uiHi) ;
          FreeDDElParam (WM_DDE_ADVISE, lParam) ;

          hwndClient = (HWND) wParam ;
          hDdeAdvise = (GLOBALHANDLE) uiLow ;
          aItem      = (ATOM) uiHi ;

          pDdeAdvise = (DDEADVISE *) GlobalLock (hDdeAdvise) ;

               // Check for matching format and data item

          if (pDdeAdvise->cfFormat == CF_TEXT)
               {
               GlobalGetAtomName (aItem, szItem, sizeof (szItem)) ;
```

```
          for (i = 0 ; i < NUM_STATES ; i++)
               if (strcmp (szItem, pop[i].szState) == 0)
                    break ;

               // Fill in the POPADVISE structure and
                    acknowledge
          if (i < NUM_STATES)
               {
               hPopAdvise = (GLOBALHANDLE) GetWindowLong (hwnd, 4) ;
               pPopAdvise = (POPADVISE *)
                                   GlobalLock (hPopAdvise) ;

               pPopAdvise[i].fAdvise    = TRUE ;
               pPopAdvise[i].fDeferUpd = pDdeAdvise->fDeferUpd ;
               pPopAdvise[i].fAckReq    = pDdeAdvise->fAckReq ;
               pPopAdvise[i].lPopPrev   = pop[i].lPop ;

               GlobalUnlock (hDdeAdvise) ;
               GlobalFree (hDdeAdvise) ;

               DdeAck.bAppReturnCode = 0 ;
               DdeAck.reserved       = 0 ;
               DdeAck.fBusy          = FALSE ;
               DdeAck.fAck           = TRUE ;

               wStatus = *((WORD *) &DdeAck) ;

               if (!PostMessage (hwndClient, WM_DDE_ACK,
                                  (WPARAM) hwnd,
                                  PackDDElParam (WM_DDE_ACK,
                                                 wStatus, aItem)))
                    {
                    GlobalDeleteAtom (aItem) ;
                    }
               else
                    {
                    PostDataMessage (hwnd, hwndClient, i,
                                     pPopAdvise[i].fDeferUpd,
                                     pPopAdvise[i].fAckReq,
                                     FALSE) ;
                    }

               GlobalUnlock (hPopAdvise) ;
               return 0 ;
               }
          }

               // Otherwise, post a negative WM_DDE_ACK

     GlobalUnlock (hDdeAdvise) ;

     DdeAck.bAppReturnCode = 0 ;
     DdeAck.reserved       = 0 ;
     DdeAck.fBusy          = FALSE ;
     DdeAck.fAck           = FALSE ;
```

```
          wStatus = *((WORD *) &DdeAck) ;

          if (!PostMessage (hwndClient, WM_DDE_ACK, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ACK, wStatus, aItem)))
               {
               GlobalFree (hDdeAdvise) ;
               GlobalDeleteAtom (aItem) ;
               }

          return 0 ;

     case WM_DDE_UNADVISE :

               // wParam          -- sending window handle
               // LOWORD (lParam) -- data format
               // HIWORD (lParam) -- item atom

          hwndClient = (HWND) wParam ;
          cfFormat   = LOWORD (lParam) ;
          aItem      = HIWORD (lParam) ;

          DdeAck.bAppReturnCode = 0 ;
          DdeAck.reserved       = 0 ;
          DdeAck.fBusy          = FALSE ;
          DdeAck.fAck           = TRUE ;

          hPopAdvise  = (GLOBALHANDLE) GetWindowLong (hwnd, 4) ;
          pPopAdvise = (POPADVISE *) GlobalLock (hPopAdvise) ;

               // Check for matching format and data item

          if (cfFormat == CF_TEXT || cfFormat == NULL)
               {
               if (aItem == (ATOM) NULL)
                    for (i = 0 ; i < NUM_STATES ; i++)
                         pPopAdvise[i].fAdvise = FALSE ;
               else
                    {
                    GlobalGetAtomName (aItem, szItem, sizeof (szItem)) ;

                    for (i = 0 ; i < NUM_STATES ; i++)
                         if (strcmp (szItem, pop[i].szState) == 0)
                              break ;

                    if (i < NUM_STATES)
                         pPopAdvise[i].fAdvise = FALSE ;
                    else
                         DdeAck.fAck = FALSE ;
                    }
               }
          else
               DdeAck.fAck = FALSE ;

               // Acknowledge either positively or negatively
```

```
          wStatus = *((WORD *) &DdeAck) ;

          if (!PostMessage (hwndClient, WM_DDE_ACK, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ACK, wStatus, aItem)))
               {
               if (aItem != (ATOM) NULL)
                    GlobalDeleteAtom (aItem) ;
               }

          GlobalUnlock (hPopAdvise) ;
          return 0 ;

     case WM_DDE_EXECUTE :

               // Post negative acknowledge

          hwndClient = (HWND) wParam ;
          hCommands  = (GLOBALHANDLE) lParam ;

          DdeAck.bAppReturnCode = 0 ;
          DdeAck.reserved       = 0 ;
          DdeAck.fBusy          = FALSE ;
          DdeAck.fAck           = FALSE ;

          wStatus = *((WORD *) &DdeAck) ;

          if (!PostMessage (hwndClient, WM_DDE_ACK, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ACK,
                                           wStatus, (UINT) hCommands)))
               {
               GlobalFree (hCommands) ;
               }
          return 0 ;

     case WM_DDE_POKE :

               // Post negative acknowledge

          UnpackDDElParam (WM_DDE_POKE, lParam, &uiLow, &uiHi) ;
          FreeDDElParam (WM_DDE_POKE, lParam) ;
          hwndClient = (HWND) wParam ;
          hDdePoke   = (GLOBALHANDLE) uiLow ;
          aItem      = (ATOM) uiHi ;

          DdeAck.bAppReturnCode = 0 ;
          DdeAck.reserved       = 0 ;
          DdeAck.fBusy          = FALSE ;
          DdeAck.fAck           = FALSE ;

          wStatus = *((WORD *) &DdeAck) ;

          if (!PostMessage (hwndClient, WM_DDE_ACK, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ACK, wStatus, aItem)))
               {
```

```
                    GlobalFree (hDdePoke) ;
                    GlobalDeleteAtom (aItem) ;
                    }

               return 0 ;

          case WM_DDE_TERMINATE :

                    // Respond with another WM_DDE_TERMINATE iMsg

               hwndClient = (HWND) wParam ;
               PostMessage (hwndClient, WM_DDE_TERMINATE, (WPARAM) hwnd, 0L) ;
               DestroyWindow (hwnd) ;
               return 0 ;

          case WM_TIMER :

                    // Post WM_DDE_DATA iMsgs for changed populations

               hwndClient = (HWND) GetWindowLong (hwnd, 0) ;
               hPopAdvise = (GLOBALHANDLE) GetWindowLong (hwnd, 4) ;
               pPopAdvise = (POPADVISE *) GlobalLock (hPopAdvise) ;

               for (i = 0 ; i < NUM_STATES ; i++)
                    if (pPopAdvise[i].fAdvise)
                         if (pPopAdvise[i].lPopPrev != pop[i].lPop)
                              {
                              if (!PostDataMessage (hwnd, hwndClient, i,
                                                    pPopAdvise[i].fDeferUpd,
                                                    pPopAdvise[i].fAckReq,
                                                    FALSE))
                                   break ;

                              pPopAdvise[i].lPopPrev = pop[i].lPop ;
                              }

               GlobalUnlock (hPopAdvise) ;
               return 0 ;

          case WM_CLOSE :

                    // Post a WM_DDE_TERMINATE iMsg to the client

               hwndClient = (HWND) GetWindowLong (hwnd, 0) ;
               PostMessage (hwndClient, WM_DDE_TERMINATE, (WPARAM) hwnd, 0L) ;

               dwTime = GetCurrentTime () ;

               while (GetCurrentTime () - dwTime < DDE_TIMEOUT)
                    if (PeekMessage (&msg, hwnd, WM_DDE_TERMINATE,
                                     WM_DDE_TERMINATE, PM_REMOVE))
                         break ;

               DestroyWindow (hwnd) ;
               return 0 ;
```

```
          case WM_DESTROY :
               hPopAdvise = (GLOBALHANDLE) GetWindowLong (hwnd, 4) ;
               GlobalFree (hPopAdvise) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK TimerEnumProc (HWND hwnd, LPARAM lParam)
     {
     SendMessage (hwnd, WM_TIMER, 0, 0L) ;

     return TRUE ;
     }

BOOL CALLBACK CloseEnumProc (HWND hwnd, LPARAM lParam)
     {
     SendMessage (hwnd, WM_CLOSE, 0, 0L) ;

     return TRUE ;
     }

BOOL PostDataMessage (HWND hwndServer, HWND hwndClient, int iState,
                      BOOL fDeferUpd, BOOL fAckReq, BOOL fResponse)
     {
     ATOM         aItem ;
     char         szPopulation[16] ;
     DDEACK       DdeAck ;
     DDEDATA     *pDdeData ;
     DWORD        dwTime ;
     GLOBALHANDLE hDdeData ;
     MSG          msg ;
     WORD         wStatus ;

     aItem = GlobalAddAtom (pop[iState].szState) ;

          // Allocate a DDEDATA structure if not deferred update

     if (fDeferUpd)
          {
          hDdeData = NULL ;
          }
     else
          {
          wsprintf (szPopulation, "%ld\r\n", pop[iState].lPop) ;

          hDdeData = GlobalAlloc (GHND * GMEM_DDESHARE,
                                  sizeof (DDEDATA) + strlen (szPopulation)) ;

          pDdeData = (DDEDATA *) GlobalLock (hDdeData) ;

          pDdeData->fResponse = fResponse ;
          pDdeData->fRelease  = TRUE ;
          pDdeData->fAckReq   = fAckReq ;
```

```
          pDdeData->cfFormat   = CF_TEXT ;

          lstrcpy ((PSTR) pDdeData->Value, szPopulation) ;

          GlobalUnlock (hDdeData) ;
          }

          // Post the WM_DDE_DATA iMsg
     if (!PostMessage (hwndClient, WM_DDE_DATA, (WPARAM) hwndServer,
                    PackDDElParam (WM_DDE_DATA, (UINT) hDdeData, aItem)))
          {
          if (hDdeData != NULL)
               GlobalFree (hDdeData) ;

          GlobalDeleteAtom (aItem) ;
          return FALSE ;
          }

          // Wait for the acknowledge iMsg if it's requested

     if (fAckReq)
          {
          DdeAck.fAck = FALSE ;

          dwTime = GetCurrentTime () ;

          while (GetCurrentTime () - dwTime < DDE_TIMEOUT)
               {
               if (PeekMessage (&msg, hwndServer, WM_DDE_ACK, WM_DDE_ACK,
                                PM_REMOVE))
                    {
                    wStatus = LOWORD (msg.lParam) ;
                    DdeAck = *((DDEACK *) &wStatus) ;
                    aItem  = HIWORD (msg.lParam) ;
                    GlobalDeleteAtom (aItem) ;
                    break ;
                    }
               }

          if (DdeAck.fAck == FALSE)
               {
               if (hDdeData != NULL)
                    GlobalFree (hDdeData) ;

               return FALSE ;
               }
          }

     return TRUE ;
     }
```

DDEPOP.H

```
/*-------------------------------------------------------------
   DDEPOP.H header file
```

```
   Data from "The World Almanac and Book of Facts 1992," page 75
  ---------------------------------------------------------------*/

#include <time.h>

struct
     {
     char        *szState ;
     long         lPop70 ;
     long         lPop80 ;
     long         lPop90 ;
     long         lPop ;
     long         lPopLast ;
     long double a ;
     long double b ;
     long double c ;
     }
     pop[] = {
              "AL",   3444354,   3894025,   4040587, 0, 0, 0.0, 0.0, 0.0,
              "AK",    302583,    401851,    550043, 0, 0, 0.0, 0.0, 0.0,
              "AZ",   1775399,   2716546,   3665228, 0, 0, 0.0, 0.0, 0.0,
              "AR",   1923322,   2286357,   2350725, 0, 0, 0.0, 0.0, 0.0,
              "CA",  19971069,  23667764,  29760021, 0, 0, 0.0, 0.0, 0.0,
              "CO",   2209596,   2889735,   3294394, 0, 0, 0.0, 0.0, 0.0,
              "CT",   3032217,   3107564,   3287116, 0, 0, 0.0, 0.0, 0.0,
              "DE",    548104,    594338,    666168, 0, 0, 0.0, 0.0, 0.0,
              "DC",    756668,    638432,    606900, 0, 0, 0.0, 0.0, 0.0,
              "FL",   6791418,   9746961,  12937926, 0, 0, 0.0, 0.0, 0.0,
              "GA",   4587930,   5462982,   6478216, 0, 0, 0.0, 0.0, 0.0,
              "HI",    769913,    964691,   1108229, 0, 0, 0.0, 0.0, 0.0,
              "ID",    713015,    944127,   1006749, 0, 0, 0.0, 0.0, 0.0,
              "IL",  11110285,  11427409,  11430602, 0, 0, 0.0, 0.0, 0.0,
              "IN",   5195392,   5490214,   5544159, 0, 0, 0.0, 0.0, 0.0,
              "IA",   2825368,   2913808,   2776755, 0, 0, 0.0, 0.0, 0.0,
              "KS",   2249071,   2364236,   2477574, 0, 0, 0.0, 0.0, 0.0,
              "KY",   3220711,   3660324,   3685296, 0, 0, 0.0, 0.0, 0.0,
              "LA",   3644637,   4206116,   4219973, 0, 0, 0.0, 0.0, 0.0,
              "ME",    993722,   1125043,   1227928, 0, 0, 0.0, 0.0, 0.0,
              "MD",   3923897,   4216933,   4781468, 0, 0, 0.0, 0.0, 0.0,
              "MA",   5689170,   5737093,   6016425, 0, 0, 0.0, 0.0, 0.0,
              "MI",   8881826,   9262044,   9295297, 0, 0, 0.0, 0.0, 0.0,
              "MN",   3806103,   4075970,   4375099, 0, 0, 0.0, 0.0, 0.0,
              "MS",   2216994,   2520770,   2573216, 0, 0, 0.0, 0.0, 0.0,
              "MO",   4677623,   4916766,   5117073, 0, 0, 0.0, 0.0, 0.0,
              "MT",    694409,    786690,    799065, 0, 0, 0.0, 0.0, 0.0,
              "NE",   1485333,   1569825,   1578385, 0, 0, 0.0, 0.0, 0.0,
              "NV",    488738,    800508,   1201833, 0, 0, 0.0, 0.0, 0.0,
              "NH",    737681,    920610,   1109252, 0, 0, 0.0, 0.0, 0.0,
              "NJ",   7171112,   7365011,   7730188, 0, 0, 0.0, 0.0, 0.0,
              "NM",   1017055,   1303302,   1515069, 0, 0, 0.0, 0.0, 0.0,
              "NY",  18241391,  17558165,  17990455, 0, 0, 0.0, 0.0, 0.0,
              "NC",   5084411,   5880095,   6628637, 0, 0, 0.0, 0.0, 0.0,
              "ND",    617792,    652717,    638800, 0, 0, 0.0, 0.0, 0.0,
              "OH",  10657423,  10797603,  10847115, 0, 0, 0.0, 0.0, 0.0,
```

```
          "OK",    2559463,   3025487,   3145585, 0, 0, 0.0, 0.0, 0.0,
          "OR",    2091533,   2633156,   2842321, 0, 0, 0.0, 0.0, 0.0,
          "PA",   11800766,  11864720,  11881643, 0, 0, 0.0, 0.0, 0.0,
          "RI",     949723,    947154,   1003464, 0, 0, 0.0, 0.0, 0.0,
          "SC",    2590713,   3120729,   3486703, 0, 0, 0.0, 0.0, 0.0,
          "SD",     666257,    690768,    696004, 0, 0, 0.0, 0.0, 0.0,
          "TN",    3926018,   4591023,   4877185, 0, 0, 0.0, 0.0, 0.0,
          "TX",   11198655,  14225513,  16986510, 0, 0, 0.0, 0.0, 0.0,
          "UT",    1059273,   1461037,   1722850, 0, 0, 0.0, 0.0, 0.0,
          "VT",     444732,    511456,    562758, 0, 0, 0.0, 0.0, 0.0,
          "VA",    4651448,   5346797,   6187358, 0, 0, 0.0, 0.0, 0.0,
          "WA",    3413244,   4132353,   4866692, 0, 0, 0.0, 0.0, 0.0,
          "WV",    1744237,   1950186,   1793477, 0, 0, 0.0, 0.0, 0.0,
          "WI",    4417821,   4705642,   4891769, 0, 0, 0.0, 0.0, 0.0,
          "WY",     332416,    469557,    453588, 0, 0, 0.0, 0.0, 0.0,
          "US",  203302031, 226542203, 248709873, 0, 0, 0.0, 0.0, 0.0
          } ;

#define NUM_STATES (sizeof (pop) / sizeof (pop[0]))

void CalcPops (void)
     {
     int    i ;
     time_t lTime ;

     time (&lTime) ;                      // time in seconds since 1/1/70
     lTime -= 92L * 24 * 60 * 60 ;        // time in seconds since 4/1/70

     for (i = 0 ; i < NUM_STATES ; i++)
          pop[i].lPop = (long) (pop[i].a * lTime * lTime +
                                pop[i].b * lTime +
                                pop[i].c) ;
     }

void InitPops (void)
     {
     int         i ;
     long double ldSec80, ldSec90 ;

     ldSec80 = 3653.0 * 24 * 60 * 60 ;  // seconds from 4/1/70 to 4/1/80
     ldSec90 = 7305.0 * 24 * 60 * 60 ;  // seconds from 4/1/70 to 4/1/90

     for (i = 0 ; i < NUM_STATES ; i++)
          {
          pop[i].a = (ldSec90 * (pop[i].lPop80 - pop[i].lPop70) +
                      ldSec80 * (pop[i].lPop70 - pop[i].lPop90)) /
                     (ldSec90 * ldSec80 * (ldSec80 - ldSec90)) ;

          pop[i].b = (ldSec90 * ldSec90 * (pop[i].lPop70 - pop[i].lPop80) +
                      ldSec80 * ldSec80 * (pop[i].lPop90 - pop[i].lPop70)) /
                     (ldSec90 * ldSec80 * (ldSec80 - ldSec90)) ;

          pop[i].c = pop[i].lPop70 ;
          }
```

```
    CalcPops () ;
    }
```

DDEPOP1.RC

```
/*----------------------------
   DDEPOP1.RC resource script
  ----------------------------*/

DdePop1 ICON ddepop.ico
```

DDEPOP.ICO

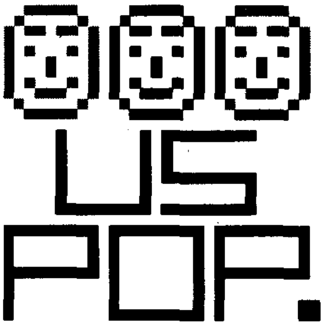


图 17-1 DDEPOP1程序

如前所述,你可以在 Microsoft Excel 使用此服务器。本章稍后将给出一个也使用 DDEPOP1 作为服务器的 DDE 客户程序 (SHOWPOP1)。

17.2.1 DDEPOP1 程序

你会注意到 DDEPOP1.C 程序开始处包含如下行:

```
#include<dde.h>
```

这是包含 DDE 消息和数据结构的头文件。

然后程序使用 typedef 语句定义一个结构(POPADVISE)。稍后我将讨论如何使用此结构。

在 WinMain 中，程序注册了两个窗口类。第一个类的名称是 “ DdePop1 ” 且作为程序的主窗口；第二个类的名称是 “ DdePop1.Server ” ，那些创建出来维护多个 DDE 对话的子窗口使用这个类，每个对话需要基于此窗口类的自己的子窗口。

在第二个窗口类中， WNDCLASS 结构的 chWndExtra 域被设置成每个窗口容纳两个字。后面将会提到，第一个字用于存储与服务器窗口进行通讯的客户窗口的句柄，第二个字是包含 POPADVISE 类型的 NUM_STATE 结构的全局内存块句柄。

在 DDEPOP1 创建了主窗口之后，它将调用 InitPops 。这个函数在 DDEPOP.H 中，该文件中还有实际的人口数据和一个叫做 CalcPops 的用来计算瞬时人口数的函数。DDEPOP1 也调用 SetTimer 来设置一个 5 秒间隔的定时器，以便周期性地更新 DDEPOP1.H 中 pop 结构的 lPop 域。

可以看到，用 SW_SHOWMINNOACTIVE 参数调用 ShowWindow 并且 WndProc 从 WM_QUERYOPEN 消息返回 0 值，这就使 DDEPOP 被显示为任务栏上的一个按钮。

17. 2. 2　WM_DDE_INITIATE 消息

客户通过向所有顶层窗口广播WM_DDE_INITIATE消息来发动一个DDE对话(在本章后面讨论DDE客户程序时你会看到，这是使用HWND_BROADCAST句柄作为第一个参数调用 SendMessage 来完成的)。

DDE 服务器在主窗口过程中处理 WM_DDE_INITIATE 消息。与其它 DDE 消息一样，wParam 参数都是发送消息的窗口的句柄，即客户的窗口句柄，WndProc 在变量 hwndClient 中保存此句柄。

对于 WM_DDE_INITIATE 消息，lParam 的低位字是标识所希望的应用程序的原子。如果客户希望得到任何服务器的响应，则它可能为 NULL 。 lParam 的高位字是标识所希望的主题的原子。如果客户希望从可提供任何主题的服务器上得到响应，则它也为 NULL 。

WndProc 通过调用 GlobalAddAtom 为其应用程序名（“ DdePop1 ”和主题名“ US_POpulation ”）增加原子来处理 WM_DDE_INITIATE 消息，然后它检查由 lParam 的低位字和高位字提供的原子是 NULL 还是与这些原子相匹配。

如果原子匹配，WndProc 创建一个基于“ DdePop1.Server ”窗口类的隐藏子窗口，这个窗口(它的窗口过程是 SeverProc)将处理 DDE 对话中所有的后续 DDE 消息。为窗口预留的两个字中的前一个字用 SetWindowWord 设置为客户句柄。然后，WndProc 通过发送一条 WM_DDE_ACK 消息给客户来响应 WM_DDE_INITIATE 消息，wParam 参数是刚创建的服务器窗口的句柄，而 lParam 包含标识服务器应用程序名和主题名的原子（如果客户请求所有的主题并且服务器支持多个主题，则服务器发送多条 WM_DDE_ACK 消息给客户，每条对应于它所支持的一个主题）。

接收WM_DDE_ACK 消息的程序负责删除伴随这条消息的所有原子。只有当WndProc 不发送 WM_DDE_ACK 消息给客户时，它才会为自己所创建的两个原子调用 GlobalDeleteAtom 。

WM_DDE_INITIATE 消息和 WM_DDE_ACK 消息(响应 WM_DDE_INITIATE)是仅有的两条由 SendMessage 发送而不是由 PostMessge 邮递的 DDE 消息。我们在本章后面将会看到，这意味着发送了 WM_DDE_INITIATE 消息的客户在发消息的 SendMessage 调用返回之前就收到了 WM_DDE_ACK 响应。

17. 2. 3　ServerProc 窗口过程

随着发送 WM_DDE_ACK 消息来响应 WM_DDE_INITIATE 消息，DDE 对话就开始了。如前所述，当 WndProc 发送 WM_DDE_ACK 消息给客户时，它将 wParam 参数设置为对话所创建的子窗口的句柄。这意味着所有后续的DDE消息将发生在客户和此子窗口之间，子窗口的窗口过程为 ServerProc 。

处理 WM_CREATE 消息时，ServerProc 分配 NUMSTATES 个 POPADVISE 类型的结

构所需要的内存(后面很快会讨论如何使用 POPADVISE)，这个内存块的句柄由 SetWindowWord 作为第二个预留字保存起来。当 ServerProc 收到 WM_DESTORY 消息时，将释放这个内存块。

17.2.4 WM_DDE_REQUEST 消息

当需要与一个特定项相关联的数据时，客户邮递一条 WM_DDE_REQUEST 消息给服务器，这就是那种冷链接方式。服务器通过邮递一条带有所请求数据的 WM_DDE_DATA 消息来响应请求，而如果服务器不能满足这个请求，则邮递一条 WM_DDE_ACK 消息。让我们看一看 ServerProc 是如何处理 WM_DDE_REQUEST 消息的。

与通常的 DDE 消息一样，伴随 WM_DDE_REQUEST 的 wParam 参数是邮递消息的窗口的句柄，此时是客户；lParam 的低位字是所要求的数据格式，lParam 的高位字是被请求数据项的原子。

DDE 数据的格式同剪贴板数据的格式一样，所以 lParam 的这个低位字通常是以 CF 前缀开头的标识符之一。客户可能为具有不同数据格式的同一项发送多条 WM_DDE_REQUEST 消息给服务器，服务器只能对它所支持的格式用 WM_DDE_DATA 消息作出响应。最普通的 DDE 数据格式是 CF_TEXT，这是 DDEPOP1 支持的唯一格式。

所以，当处理 WM_DDE_REQUEST 消息时，ServerProc 首先检查所请求的格式是否为 CF_TEXT，然后，ServerProc 调用 GlobalGetAtomName 函数来得到与 lParam 高位字传送的原子相关联的字符串。如果客户知道它正在做些什么，那么它将是标识州名的双字符串。由一个 for 循环检查这个州名并尝试用 pop 结构的 szState 域去匹配它。如果匹配，ServerProc 就调用 GlobalDeleteAtom 来删除这个原子，然后调用 PostDataMessage(这是在 DDEPOP1 尾部用于邮递 WM_DDE_DATA 消息的函数，稍后将讨论它)，最后 ServerProc 返回。

如果所请求的格式不是 CF_TEXT，或者不存在与州名匹配的项原子，则 SenrverProc 邮递一个否定的 WM_DDE_ACK 消息表明不能提供数据。这是通过设置 DDEACK 结构(在 DDE.H 中定义)中的 fAck 域为 FALSE 来实现的。DDEACK 结构被转化为一个字，它形成 lParam 的低位字。lParam 的高位字是被请求项的原子。PostMessage 把 WM_DDE_ACK 消息邮递给客户。

注意这里是如何处理原子的。WM_DDE_REQUEST 的文档提到：“当用 WM_DDE_DATA 消息响应时，可以使用 aItem 原子或者也可以删除它并创建一个新原子”。意思是说，全局原子表的状态不应该被服务器改变——亦即，aItem 原子的引用计数不应该增加或减少。

这里存在三种情况：

- 如果请求格式为 CF_TEXT 并且原子同州名之一匹配，则 ServerProc 在调用 DDEPOP1.C 中名为 PostDataMessage 的函数之前调用 GlobalDeleteAtom 。当给客户发送 WM_DDE_DATA 消息时，这个 PostDataMessage 函数（稍后将会看到）重建这个原子。
- 如果请求格式不是 CF_TEXT ，或者原子不与任何州名匹配，则 ServerProc 调用 PostMessage 发送一个否定的 WM_DDE_ACK 消息给客户，该消息简单地重用此

原子。

- 然而，如果这个 PostMessage 调用失败（也许客户意外地终止了），则 ServerProc 删除这个原子，因为客户已经不能删除它了。

我们还没有讨论完 WM_DDE_REQUEST 消息，因为还没有查看 DDEPOP1 的 PostDataMessage 如何用 WM_DDE_DATA 消息作出响应。这正是下面将要讨论的内容。

17.2.5 DDEPOP1 的 PostDataMessage 函数

DDEPOP1.C 结尾处的 PostDataMessage 函数负责向客户发送 WM_DDE_DATA 消息。这个函数也负责处理 WM_DDE_ADVISE 消息（稍后将讨论），所以它比只处理 WM_DDE_REQUEST 消息更复杂一些。

PostDataMessage 有六个参数：

- hwndServer——服务器窗口句柄
- hwndClient——客户窗口句柄
- iState——它是识别需要哪个州人口数据的 pop 数组的下标
- fDeferUpd——响应 WM_DDE_REQUEST 消息时被 ServerProc 设置为 FALSE
- fAckReq——在这种情况下也被 ServerProc 设置为 FALSE
- fResponse——被 ServerProc 设置为 TRUE 以指示一个对 WM_DDE_REQUEST 的响应

（当我们涉及 WM_DDE_ADVISE 消息时将讨论 fDeferUpd 和 fAckReq 参数，这两个参数中的任何一个被设置为 TRUE 时，将忽略 PostDataMessage 的所有部分。）

PostDataMessage 一开始就调用 GlobalAddItem 为双字符的州名创建一个原子（注意在调用 PostDataMessage 之前，ServerProc 删除这个原子），然后调用 wsprintf，将州人口数转换（由 WndProc 每隔 5 秒更新一次）为回车符和换行符结尾的字符串。

然后，PostDataMessage 以 GMEM_DDESHARE 选项调用 GlobalAllco 以分配一块内存，在此保存 DDEDATA 结构（在 DDE.H 中定义）以及紧随其后的实际数据（字符串 szPopulation）。在开始使用 PostDataMessage 响应 WM_DDE_REQUEST 消息时，DDEDATA 结构被如下进行设置：

- DDEDATA 的 fResponse 域被设置为 TRUE，指明数据是对 WM_DDE_DATA 消息的响应。
- fRelease 域也被设置为 TRUE，指明客户应该释放刚分配的内存块。
- fAckReq 域被设置为 FALSE，指明不需要来自客户的 WM_DDE_ACK 消息。
- cfFormat 域被设置为 CF_TEXT，指明数据是文本格式。
- szPopulation 数组被拷贝到从结构的 Value 域开始的内存块区域中。

PostDataMessage 然后使用 PostMessage 给客户发送一条 WM_DDE_DATA 消息。通常情况下，wParam 是发送消息的窗口（服务器）句柄。lParam 参数要复杂一些，它标识包含两个值的一个内存对象，这两个值分别为：包含 DDEDATA 结构的内存块句柄和数据项（双字符州名）的原子。lParam 中的标识符是通过调用 PackDDElParam 来创建的。接受 WM_DDE_DATA 消息的过程必须通过调用 UnpackDDElParam，从 lParam 内存对象中抽

取这两个数据项，然后通过调用 FreeDDElParam 来释放对象。

如果 PostMessage 成功，则工作即告结束。客户负责释放内存块和删除原子。如果 PostMessage 失败（可能是客户意外终止），那么 PostDataMessage 就释放所分配的内存块并删除原子。

17.2.6 WM_DDE_ADVISE 消息

至此，你可能已经开始意识到 DDE 的某些复杂性。使用 WM_DDE_ADVISE 和热链接还要稍微复杂些。

刚才讨论的 WM_DDE_REQUEST 消息允许客户从服务器得到数据，但是如果数据会发生变化(瞬时人口数也会变化)，那么客户就没有办法知道这一点。在数据被更新时通知客户是 WM_DDE_ADVISE 消息的目的。在接收这条消息时，服务器负责在数据发生变化时通知客户(这是由服务器给客户邮递 WM_DDE_DATA 消息来完成的)。这可能有些复杂，因为服务器必须“记住”客户请求对那些项的更新通知；而且，客户还要求以特定的方式邮递此数据。

在 WM_DDE_ADVISE 消息中，lParam 包含在 DDE.H 中定义的 DDEADVISE 结构的全局内存句柄，同时还含有标识数据项的原子。处理 WM_DDE_ADVISE 消息时，ServerProc 首先通过调用 UnpackDDElParam 抽出句柄和原子，然后调用 FreeDDElParam 释放 lParam 对象。接着，它检查 DDEADVISE 结构中的 cfFormat 域是否为 CF_TEXT，如果是，那么它可以得到原子所引用的文本串并与 pop 结构的 szState 域核对。

如果这两者匹配，则 ServerProc 得到一个在 WM_CREATE 消息期间分配的指向 POPADVISE 结构串的指针。每个州对应于数组中的一个 POPADVISE 结构，而每个进行 DDE 对话的窗口都有不同的数组。这个数组被用来存放 ServerProc 为客户更新数据项所需的所有信息。

选定州的 POPADVISE 结构域设置如下:

- fAdvise 被设置为 TRUE 。这是表明客户希望得到更新后的此州数据的标志。
- fDeferUpd(“延迟更新”)域设置成 DDEADVISSE 结构中相同域的值。TRUE 值指客户希望创建温链接而不是热链接。此后，客户在得到数据之前先被告知数据发生了变化，而不是立刻获得数据（这种情况下，服务器以 NULL 值而不是包含 DDEDATA 结构的全局内存块句柄来邮递一条 WM_DDE_DATA 消息。然后，客户将邮递一条常规的 WM_DDE_REQUEST 消息来得到实际数据）。FALSE 值表示客户需要 WM_DDE_DATA 消息中的数据。
- fAckReq (“需要确认”)域被设置成 DDEADVISE 结构中相同域的值。这个值的含义有些复杂。TRUE 值指示服务器在邮递 WM_DDE_DATA 消息时把 DDEDATA 结构的 fAckReq 设置为 TRUE，以便要求客户用 WM_DDE_ACK 消息来确认 WM_DDE_DATA 消息。TRUE 值并不表示客户要求来自服务器的 WM_DDE_ACK 消息；而是要求服务器在以后邮递 WM_DDE_DATA 消息时，向客户请求 WM_DDE_ACK 消息。
- lPopPrev 域设置为州的当前人口数。ServerProc 使用这个域确定客户是否需要人

口数发生改变的通知。

至此，ServerProc 不再需要 DDEAADVISE 结构，并且遵循文档上关于 WM_DDE_ADVISE 的建议释放内存块。ServerProc 必须邮递肯定的 WM_DDE_ACK 消息来响应 WM_DDE_ADVISE 消息，DDEACK 结构的 fAck 域被设置为 TRUE 。如果 PostMessage 失败，则 ServerProc 删除这个原子并释放 DDEADVISE 内存块。

如果数据格式不是 CF_TEXT，或者州名不匹配，则 ServerProc 邮递一条否定的 WM_DDE_ACK 消息。在这种情况下，如果 PostMessage 调用失败，则 ServerProc 既删除原子，也要释放 DDEADVISE 内存块。

从理论上讲，对 WM_DDE_ADVISE 消息的处理是完成了。然而，客户要求一旦数据项发生变化就要接到通知。假定客户不知道数据项的任何值，那么 ServerProc 就有必要向客户邮递一条 WM_DDE_DATA 消息。

这就要使用 PostDataMessage 函数，并把它的第三个参数设置为 POPADVISE 结构的 fDeferUpd 域，第四个参数设置为 POPADVISE 结构的 fAckReq 域，而最后一个参数设置为 FALSE(指定邮递的 WM_DDE_DATA 消息是对 WM_DDE_ADVISE 的响应，而不是对 WM_DDE_REQUEST 的响应)。

是再一次讨论 PostDaraMessage 的时候了。在函数的开始，如果 fDefUpd 参数为 TRUE，则函数简单地把 hDdeData 设置为 NULL 而不给它分配内存。

如果 fAckReq 参数为 TRUE，则 PostDataMessage 在邮递 WM_DDE_DATA 消息之后等待来自客户的 WM_DDE_ACK 消息。它是通过调用 PeekMessage 达到这一目的的。PostDataMessage 删除 WM_DDE_ACK 消息中的原子。如果 WM_DDE_ACK 消息没有在 3 秒之内到达——或者如果此消息是否定确认——则 PostDataMessage 释放包含 DDE_DATA 结构的数据块。

如果你以为可以假定客户邮递 WM_DDE_ADVISE 消息时从来不会将延迟更新或者确认请求位设置为 TRUE，从而略过这部分工作，那么你得再猜一次了。Microsoft Excel 恰恰是这样做的，它建立了需要对 WM_DDE_DATA 消息进行确认的温链接。

17.2.7　更新数据项

处理完 WM_DDE_ADVISE 消息之后，服务器应该在数据项发生变化时通知客户。在 DDEPOP1 中，使用定时器每隔 5 秒钟重新计算一次人口数，这发生在 WndProc 对 WM_TIMER 消息的处理时。

然后 WndProc 用 TimerEnumProc 函数(在 DDEPOP1.C 中的 ServerProc 之后)调用 EnumChildWindows 。TimerEnumProc 发送 WM_TIMER 消息给所有子窗口，这些子窗口均使用 ServerProc 窗口过程。

ServerProc 函数处理 WM_TIMER 消息，循环处理各个州，并检查 POPADVISE 结构的 fAdvise 域是否被设置为 TRUE 以及人口数是否发生了变化。如果是这样，就调用 PostDataMessage 给客户邮递 WM_DDE_DATA 消息。

17.2.8 WM_DDE_UNADVISE 消息

WM_DDE_UNADVISE 消息指示服务器停止在数据项发生变化时邮递 WM_DDE_DATA 消息。lParam 的低位字为数据格式或者为 0，后者代表所有数据格式。lParam 的高位字为 ATOM 或 NULL，后者代表所有项。

DDEPOP1 通过把 POPADVISE 结构中指定的 fAdvise 域设置为 FALSE 来处理 WM_DDE_UNADVISE 消息，然后用肯定或否定的 EM_DDE_ACK 消息来确认。

17.2.9 WM_DDE_TERMINATE 消息

当客户希望结束对话时，它邮递一条 WM_DDE_TERMINARE 消息给服务器，服务器简单地把自己的 WM_DDE_TERMINATE 消息发回子窗口来进行响应。在收到 WM_DDE_TERMINATE 后， ServerProc 还清除子窗口，因为不再需要它了，而且窗口所维护的对话也被终止。

ServerProc 也处理 WM_DDE_POKE 和 WM_DDE_EXECUTE 消息，但对这两条消息都简单地使用否定确认作出响应。

如果 DDEPOP1 从自己的系统菜单上关闭，则它必须终止与客户间的所有 DDE 对话。所以，当 WndProc 收到一条 WM_CLOSE 消息时，它用 CloseEnumProc 函数调用 EnumChildWindows 。 CloseEnumProc 给所有子窗口发送 WM_CLOSE 消息。

ServerProc 通过向客户邮递一条 WM_DDE_TERMINATE 消息来响应 WM_CLOSE，然后等待客户返回另一条 WM_DDE_TERMINATE 消息。

17.3 一个 DDE 客户程序

现在我们已经看到了用于 Microsoft Excel 的 DDE 服务器程序，现在再来看看使用 DDEPOP1 作为服务器的 DDE 客户程序。这个程序叫做 SHOWPOP1，如图 17-2 所示。

SHOWPOP1.MAK

```
#------------------------
# SHOWPOP1.MAK make file
#------------------------

showpop1.exe : showpop1.obj
     $(LINKER) $(GUIFLAGS) -OUT:showpop1.exe showpop1.obj $(GUILIBS)

showpop1.obj : showpop1.c showpop.h
     $(CC) $(CFLAGS) showpop1.c
```

SHOWPOP1.C

```
/*-------------------------------------------
   SHOWPOP1.C -- DDE Client using DDEPOP1
                 (c) Charles Petzold, 1996
  -------------------------------------------*/

#include <windows.h>
#include <dde.h>
#include <stdlib.h>
#include <string.h>
#include "showpop.h"

#define WM_USER_INITIATE (WM_USER + 1)
#define DDE_TIMEOUT      3000

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[] = "ShowPop1" ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "DDE Client - US Population",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     SendMessage (hwnd, WM_USER_INITIATE, 0, 0L) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

```
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static BOOL   fDoingInitiate = TRUE ;
     static char   szServerApp[]  = "DdePop1",
                   szTopic[]      = "US_Population" ;
     static HWND   hwndServer = NULL ;
     static long   cxChar, cyChar ;
     ATOM          aApp, aTop, aItem ;
     char          szBuffer[24], szPopulation[16], szItem[16] ;
     DDEACK        DdeAck ;
     DDEDATA      *pDdeData ;
     DDEADVISE    *pDdeAdvise ;
     DWORD         dwTime ;
     GLOBALHANDLE  hDdeAdvise, hDdeData ;
     HDC           hdc ;
     MSG           msg ;
     PAINTSTRUCT   ps ;
     short         i ;
     long          x, y ;
     TEXTMETRIC    tm ;
     WORD          wStatus ;
     UINT          uiLow, uiHi ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
               ReleaseDC (hwnd, hdc) ;
               return 0 ;

          case WM_USER_INITIATE :

                    // Broadcast WM_DDE_INITIATE iMsg

               aApp = GlobalAddAtom (szServerApp) ;
               aTop = GlobalAddAtom (szTopic) ;

               SendMessage (HWND_BROADCAST, WM_DDE_INITIATE, (WPARAM) hwnd,
                            MAKELONG (aApp, aTop)) ;

                    // If no response, try loading DDEPOP first

               if (hwndServer == NULL)
                    {
                    WinExec (szServerApp, SW_SHOWMINNOACTIVE) ;

                    SendMessage (HWND_BROADCAST, WM_DDE_INITIATE, (WPARAM) hwnd,
                                 MAKELONG (aApp, aTop)) ;
                    }
```

```
          // Delete the atoms

     GlobalDeleteAtom (aApp) ;
     GlobalDeleteAtom (aTop) ;
     fDoingInitiate = FALSE ;

          // If still no response, display message box

     if (hwndServer == NULL)
          {
          MessageBox (hwnd, "Cannot connect with DDEPOP1.EXE!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK) ;

          return 0 ;
          }

          // Post WM_DDE_ADVISE iMsgs

     for (i = 0 ; i < NUM_STATES ; i++)
          {
          hDdeAdvise = GlobalAlloc (GHND * GMEM_DDESHARE,
                                    sizeof (DDEADVISE)) ;

          pDdeAdvise = (DDEADVISE *) GlobalLock (hDdeAdvise) ;

          pDdeAdvise->fAckReq    = TRUE ;
          pDdeAdvise->fDeferUpd  = FALSE ;
          pDdeAdvise->cfFormat   = CF_TEXT ;

          GlobalUnlock (hDdeAdvise) ;

          aItem = GlobalAddAtom (pop[i].szAbb) ;

          if (!PostMessage (hwndServer, WM_DDE_ADVISE, (WPARAM) hwnd,
                            PackDDElParam (WM_DDE_ADVISE,
                                           (UINT) hDdeAdvise, aItem)))
               {
               GlobalFree (hDdeAdvise) ;
               GlobalDeleteAtom (aItem) ;
               break ;
               }

          DdeAck.fAck = FALSE ;

          dwTime = GetCurrentTime () ;

          while (GetCurrentTime () - dwTime < DDE_TIMEOUT)
               {
               if (PeekMessage (&msg, hwnd, WM_DDE_ACK,
                                WM_DDE_ACK, PM_REMOVE))
                    {
                    GlobalDeleteAtom (HIWORD (msg.lParam)) ;

                    wStatus = LOWORD (msg.lParam) ;
```

```
                    DdeAck = *((DDEACK *) &wStatus) ;

                    if (DdeAck.fAck == FALSE)
                        GlobalFree (hDdeAdvise) ;

                    break ;
                    }
                }

            if (DdeAck.fAck == FALSE)
                break ;

            while (PeekMessage (&msg, hwnd, WM_DDE_FIRST,
                                WM_DDE_LAST, PM_REMOVE))
                {
                DispatchMessage (&msg) ;
                }
            }

        if (i < NUM_STATES)
            {
            MessageBox (hwnd, "Failure on WM_DDE_ADVISE!",
                        szAppName, MB_ICONEXCLAMATION | MB_OK) ;
            }
        return 0 ;

    case WM_DDE_ACK :

            // In response to WM_DDE_INITIATE, save server window

        if (fDoingInitiate)
            {
            UnpackDDElParam (WM_DDE_ACK, lParam, &uiLow, &uiHi) ;
            FreeDDElParam (WM_DDE_ACK, lParam) ;
            hwndServer = (HWND) wParam ;
            GlobalDeleteAtom ((ATOM) uiLow) ;
            GlobalDeleteAtom ((ATOM) uiHi) ;
            }
        return 0 ;

    case WM_DDE_DATA :

            // wParam -- sending window handle
            // lParam -- DDEDATA memory handle & item atom

        UnpackDDElParam (WM_DDE_DATA, lParam, &uiLow, &uiHi) ;
        FreeDDElParam (WM_DDE_DATA, lParam) ;

        hDdeData  = (GLOBALHANDLE) uiLow ;
        pDdeData = (DDEDATA *) GlobalLock (hDdeData) ;
        aItem     = (ATOM) uiHi ;

            // Initialize DdeAck structure

        DdeAck.bAppReturnCode = 0 ;
```

```
DdeAck.reserved       = 0 ;
DdeAck.fBusy          = FALSE ;
DdeAck.fAck           = FALSE ;

     // Check for matching format and data item

if (pDdeData->cfFormat == CF_TEXT)
     {
     GlobalGetAtomName (aItem, szItem, sizeof (szItem)) ;

     for (i = 0 ; i < NUM_STATES ; i++)
          if (strcmp (szItem, pop[i].szAbb) == 0)
               break ;

     if (i < NUM_STATES)
          {
          strcpy (szPopulation, (char *) pDdeData->Value) ;
          pop[i].lPop = atol (szPopulation) ;
          InvalidateRect (hwnd, NULL, FALSE) ;

          DdeAck.fAck = TRUE ;
          }
     }

     // Acknowledge if necessary

if (pDdeData->fAckReq == TRUE)
     {
     wStatus = *((WORD *) &DdeAck) ;

     if (!PostMessage ((HWND) wParam, WM_DDE_ACK, (WPARAM) hwnd,
                       PackDDElParam (WM_DDE_ACK,
                                      wStatus, aItem)))
          {
          GlobalDeleteAtom (aItem) ;
          GlobalUnlock (hDdeData) ;
          GlobalFree (hDdeData) ;
          return 0 ;
          }
     }
else
     {
     GlobalDeleteAtom (aItem) ;
     }

     // Clean up

if (pDdeData->fRelease == TRUE || DdeAck.fAck == FALSE)
     {
     GlobalUnlock (hDdeData) ;
     GlobalFree (hDdeData) ;
     }
else
     {
     GlobalUnlock (hDdeData) ;
```

```
                    }

               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (i = 0 ; i < NUM_STATES ; i++)
                    {
                    if (i < (NUM_STATES + 1) / 2)
                         {
                         x = cxChar ;
                         y = i * cyChar ;
                         }
                    else
                         {
                         x = 44 * cxChar ;
                         y = (i - (NUM_STATES + 1) / 2) * cyChar ;
                         }

                    TextOut (hdc, x, y, szBuffer,
                             wsprintf (szBuffer, "%-20s",
                                       (PSTR) pop[i].szState)) ;

                    x += 36 * cxChar ;

                    SetTextAlign (hdc, TA_RIGHT * TA_TOP) ;

                    TextOut (hdc, x, y, szBuffer,
                             wsprintf (szBuffer, "%10ld", pop[i].lPop)) ;

                    SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                    }

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DDE_TERMINATE :

                    // Respond with another WM_DDE_TERMINATE iMsg

               PostMessage (hwndServer, WM_DDE_TERMINATE, (WPARAM) hwnd, 0L) ;
               hwndServer = NULL ;
               return 0 ;

          case WM_CLOSE :
               if (hwndServer == NULL)
                    break ;

                    // Post WM_DDE_UNADVISE iMsg

               PostMessage (hwndServer, WM_DDE_UNADVISE, (WPARAM) hwnd,
                            MAKELONG (CF_TEXT, NULL)) ;

               dwTime = GetCurrentTime () ;
```

```
               while (GetCurrentTime () - dwTime < DDE_TIMEOUT)
                    {
                    if (PeekMessage (&msg, hwnd, WM_DDE_ACK,
                                     WM_DDE_ACK, PM_REMOVE))
                         break ;
                    }

                    // Post WM_DDE_TERMINATE iMsg

               PostMessage (hwndServer, WM_DDE_TERMINATE, (WPARAM) hwnd, 0L) ;

               dwTime = GetCurrentTime () ;

               while (GetCurrentTime () - dwTime < DDE_TIMEOUT)
                    {
                    if (PeekMessage (&msg, hwnd, WM_DDE_TERMINATE,
                                     WM_DDE_TERMINATE, PM_REMOVE))
                         break ;
                    }

               break ;             // for default processing

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

SHOWPOP.H

```
/*-----------------------
   SHOWPOP.H header file
  -----------------------*/

struct
     {
     char *szAbb ;
     char *szState ;
     long  lPop ;
     }
     pop[] = {
             "AL", "Alabama",              0, "AK", "Alaska",             0,
             "AZ", "Arizona",              0, "AR", "Arkansas",           0,
             "CA", "California",           0, "CO", "Colorado",           0,
             "CT", "Connecticut",          0, "DE", "Delaware",           0,
             "DC", "Dist. of Columbia",    0, "FL", "Florida",            0,
             "GA", "Georgia",              0, "HI", "Hawaii",             0,
             "ID", "Idaho",                0, "IL", "Illinois",           0,
             "IN", "Indiana",              0, "IA", "Iowa",               0,
             "KS", "Kansas",               0, "KY", "Kentucky",           0,
             "LA", "Louisiana",            0, "ME", "Maine",              0,
             "MD", "Maryland",             0, "MA", "Massachusetts",      0,
             "MI", "Michigan",             0, "MN", "Minnesota",          0,
```

```
          "MS",  "Mississippi",        0, "MO",  "Missouri",            0,
          "MT",  "Montana",            0, "NE",  "Nebraska",            0,
          "NV",  "Nevada",             0, "NH",  "New Hampshire",       0,
          "NJ",  "New Jersey",         0, "NM",  "New Mexico",          0,
          "NY",  "New York",           0, "NC",  "North Carolina",      0,
          "ND",  "North Dakota",       0, "OH",  "Ohio",                0,
          "OK",  "Oklahoma",           0, "OR",  "Oregon",              0,
          "PA",  "Pennsylvania",       0, "RI",  "Rhode Island",        0,
          "SC",  "South Carolina",     0, "SD",  "South Dakota",        0,
          "TN",  "Tennessee",          0, "TX",  "Texas",               0,
          "UT",  "Utah",               0, "VT",  "Vermont",             0,
          "VA",  "Virginia",           0, "WA",  "Washington",          0,
          "WV",  "West Virginia",      0, "WI",  "Wisconsin",           0,
          "WY",  "Wyoming",            0, "US",  "United States Total", 0
          } ;
#define NUM_STATES (sizeof (pop) / sizeof (pop[0]))
```

图 17-2 SHOWPOP1程序

该程序在其窗口中显示州名和更新后的人口数据，后者是用 WM_DDE_ADVISE 从 DDEPOP1 得到的。可以看到，SHOWPOP1 中的 pop 结构同 DDEPOP1 中的结构类似，但是这个版本的 pop 结构中包含两个字母的州名缩写、州名和 lPop 域(被初始化为零)，lPop 中包含从 DDEPOP1 中得到的更新后的人口数。

SHOWPOP1 只进行一种 DDE 对话，所以它只需要一个窗口，并且将 WndProc 用于此目的。

17.3.1 发动 DDE 对话

这里，通过在 WinMain 中的 UpdateWindow 调用之后发送一条用户定义的消息(WM_USER_INITIATE)来发动 DDE 对话。通常客户可以使用菜单命令来发动对话。

作为对这条用户定义的消息的响应，WndProc 调用 GlobalAddAtom 来为服务器应用程序名(“ DdePop1 ”)和主题名(“ US_Population ”)创建原子。WindProc 通过使用一个 HWND_BROADCAST 窗口句柄调用 SendMessage 来广播 WM_DDE_INITIATE 消息。

如前如述，当应用程序名和主题名在服务器中匹配成功，该服务器就需要给客户回发一条 WM_DDE_ACK 消息。因为这条消息是用 SendMessage 发送而不是邮递的，客户会在原来以 WM_DDE_INITIATE 对 SendMessage 的调用返回之前收到 WM_DDE_ACK 消息。WndProc 通过将服务器的窗口句柄保存在变量 hwndServer 中并删除伴随这条消息的原子来处理 WM_DDE_ACK 消息。

如果客户使用 NULL 应用程序名或主题名广播一条 WM_DDE_INITIATE 消息，它必须预备接收来自满足要求的每个服务器的多条 WM_DDE_ACK 消息。这种情况下，客户必须确定使用哪个服务器。客户必须向所有其它服务器邮递 WM_DDE_TERMINATE 消息以终止对话。

在以 WM_DDE_INITIATE 调用 SendMessage 之后，hwndServer 仍有可能为 NULL 。这意味着 DDEPOP1 尚未在 Windows 下运行起来。在这种情况下，WndProc 调用 WinExec 来执行 DDEPOP1 ，WndExec 搜寻当前目录或 PATH 环境变量来加载 DDEPOP1 ，然后

WndProc 再一次广播 WM_DDE_INITIATE 消息。如果 hwndServer 仍是 NULL，则 WndProc 显示一个消息框把问题告诉客户。

下一步，对于 pop 结构中的每个州，WndProc 用 GMEM_DDESHARE 标志调用 GlobalAlloc 来分配一个 DDE_ADVISE 结构。fAckReq(“请求进行确认”)标志被设置为 TRUE(表明服务器在邮递 WM_DDE_DATA 消息时，应该将 DDEDATA 域中的 fAckReq 域设置为 NULL)，fDeferUpd 标志被设置为 FALSE (表明是热链接而不是温链接)，cfFORMAT 域被设置为 CF_TEXT，GlobalAddAtom 为双字母的州名缩写增加一个原子。

当 SHOWPOP1 邮递 WM_DDE_ADVISE 消息时，这个结构和原子被传给服务器。如果 PostMessage 调用失败（如果 DDEPOP1 意外终止就可能发生），那么 SHOWPOP1 释放内存块，删除原子并且退出循环。否则，SHOWPOP1 通过调用 PeekMessage 等待 WM_DDE_ACK 消息。正如 DDE 文档中描述的一样，如果客户使用否定确认作为响应，那么客户删除伴随该消息的原子，同时释放全局内存块。

来自客户的 WM_DDE_ACK 消息后面很可能跟有 WM_DDE_DATA 消息，因此，SHOWPOP1 调用 PeekMessage 和 DispatchMessage 从消息队列中抽取任何 DDE 消息并把它们分发给 WndProc 。

17.3.2 WM_DDE_DATA 消息

在 WM_DDE_ADVISE 消息之后，WndProc 将从服务器接收含有已更新人口数据的 WM_DDE_DATA 消息。lParam 是一个内存对象，其中包含 WM_DDE_DATA 结构的全局内存块句柄和标识数据项的原子。这些信息通过 UnpackDDElParam 提取出来。

SHOWPOP1 检查 DDEDATA 结构的 cfFormat 域是否为 CF_TEXT(当然，DDEPOP1 只能使用 CF_TEXT，不过，为了完整起见而列出了这一步)，然后调用 GlobalGetItemName 得到同这个原子项相关联的字符串。这个字符串是双字母州名的缩写。

借助 for 循环，SHOWPOP1 扫描所有州名进行匹配。如果匹配成功，则把这个州的人口数据从 DDEDATA 结构拷贝到 szPopulation 数组中，使用 C 函数 atol(“ ASCII to long ”) 转换为长整数，将其保存到 pop 结构中并使窗口无效。

现在，只剩下结束处理了。如果客户要求确认 WM_DDE_DATA 消息，WndProc 就邮递之。如果不要求确认（或 PostMessage 调用失败），则 aItem 原子被删除。如果 PostMessage 调用失败或州名不匹配(需要一个否定确认)或者 DDEDATA 结构中的 fRelease 标志被设置为 TRUE，则 SHOWPOP1 释放内存块。

在原来的 SHOWPOP1 程序中，DDEADVISE 结构的 fAckReq 域被设置为 FALSE，且邮递一条 WM_DDE_ADVISE 消息。这指示服务器把 DDEDATA 结构的 fAckReq 域设置为 FALSE 并邮递 WM_DDE_ADVISE 消息，也指示客户不应给服务器邮递 WM_DDE_ACK 消息来确认 WM_DDE_DATA 消息。这适用于正常的更新。然而，万一 SHOWPOP1 正在运行时 Windows 改变了系统时钟，则 DDEPOP1 将会邮递 52 条 WM_DDE_DATA 消息给 SHOWPOP1 而不等待确认。这将使 SHOWPOP1 的消息队列溢出而丢失许多已更新的人口数据。

教训是清楚的：如果客户希望多个可能同时发生变化的数据项，则必须把 DDEADVISE

结构的 fAckReq 域设置为 TRUE 。这是唯一安全的方法。

17.3.3 WM_DDE_TERMINATE 消息

处理服务器邮递的 WM_DDE_TERMINATE 消息是简单的： SHOWPOP1 简单地邮递另一条 WM_DDE_TERIMINATE 消息给客户，并将 hwndServer 变量设置为 NULL (说明对话结束)。

如果关闭 SHOWPOP1(由 WM_CLOSE 消息指出)，则程序首先邮递一条 WM_DDE_UNADVISE 消息给服务器，以防止任何新的数据更新。在这里，使用 NULL 原子项来指定所有数据项。然后， SHOWPOP1 邮递 WM_DDE_TERMINATE 消息给服务器并等待服务器返回 WM_DDE_TERMINATE 消息。

17.4 DDE 管 理 库

由于许多程序员在使用DDE时都感到很复杂，因此，Microsoft公司决定从Windows 3.1起赋予它新的面貌。 DDE 管理库(DDEML)通过把消息、原子管理和内存管理封装到一个函数调用接口中，从而消除了许多复杂问题。

DDEML 开始看上去可能会让人望而生畏。 DDEML.H 头文件定义了以 Dde 为前缀的 30 个函数调用。使用 DDEML 的程序还需要一个能够处理 16 种事务(transaction)类型的回调函数，这 16 种类型在 DDEML.H 中定义为以 XTYP 为前缀的常量。

DDEML 简化了 DDE 编程，从 DDEPOP1.C 和 SHOWPOP2.C 的源代码文件的长度上可以明显看出。这两个文件在功能上等价于 DDEPOP1 和 SHOWPOP1 。

17.4.1 概念上的差异

使用经典的 DDE 和使用 DDEML 存在着术语和概念方面的差异。首先，尽管所有 DDEML 事务仍然基于应用程序、主题和数据项名，但在 DDEML 中应用程序名被称为“服务”名。

使用 DDEML 的任何程序均需调用 DdeInitialize 在 DDEML 中注册自己。该函数得到一个应用程序实例识别符，它被用于所有其它 DDEML 函数。当程序结束时，调用 DdeUninitialize 。

经典 DDE 和 DDEML 之间的最显著的区别是程序用回调函数而不是用窗口过程处理 DDE 交互。该回调函数由系统调用 DdeInitialize 来注册。

在应用程序间传递数据时，程序不直接分配共享的全局内存，而是通过调用 DdeCreateDataHandle 从包含数据的缓冲区创建数据句柄。接收数据的程序通过该句柄，利用 DdeAccessData 或 DdeGeTdata 函数来获得数据。调用 DdeFreeDataHandle 即可释放数据句柄。

DDEML 程序使用串句柄来代替原子，前者存放在 HSZ 类型的变量中。

DdeCreateStringHandle 用字符串来创建串句柄，DdeQueryString 从句柄得到字符串，DdeCmpStringHandles 比较两个串句柄表示的字符串，DdeFreeStringHandle 释放串句柄。在某些情况下，串句柄被自动释放。程序通过调用 DdeKeepStringHandle 保持串句柄有效。

17.4.2 DDE 的 DDEML 方式

让我们看一看在实际的应用程序中 DDEML 是如何工作的。图 17-3 是一个命名为 DDEPOP2 的 DDEML 服务器的源代码，图 17-4 是命名为 SHOWPOP2 的 DDEML 客户的源代码。DDEPOP2 程序也需要图 17-1 中的 DDEPOP1.H 和 DDEPOP.ICO 文件，而 SHOWPOP2 程序需要图 17-2 中的 SHOWPOP.H 文件。

DDEPOP2.MAK

```
#----------------------
# DDEPOP2.MAK make file
#----------------------

ddepop2.exe : ddepop2.obj ddepop2.res
     $(LINKER) $(GUIFLAGS) -OUT:ddepop2.exe ddepop2.obj ddepop2.res $(GUILIBS)

ddepop2.obj : ddepop2.c ddepop.h
     $(CC) $(CFLAGS) ddepop2.c

ddepop2.res : ddepop2.rc ddepop.ico
     $(RC) $(RCVARS) ddepop2.rc
```

DDEPOP2.C

```
/*-----------------------------------------------
   DDEPOP2.C -- DDEML Server for Population Data
                (c) Charles Petzold, 1996
  -----------------------------------------------*/
#include <windows.h>
#include <ddeml.h>
#include <string.h>
#include "ddepop.h"

#define WM_USER_INITIATE (WM_USER + 1)
#define ID_TIMER         1

LRESULT  CALLBACK WndProc     (HWND, UINT, WPARAM, LPARAM) ;
HDDEDATA CALLBACK DdeCallback (UINT, UINT, HCONV, HSZ, HSZ,
                                          HDDEDATA, DWORD, DWORD) ;

char      szAppName[] = "DdePop2" ;
char      szTopic[]   = "US_Population" ;
DWORD     idInst ;
HINSTANCE hInst ;
HWND      hwnd ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                PSTR szCmdLine, int iCmdShow)
     {
     MSG        msg ;
     WNDCLASSEX wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = 0 ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "DDEML Population Server",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, SW_SHOWMINNOACTIVE) ;
     UpdateWindow (hwnd) ;

               // Initialize for using DDEML

     if (DdeInitialize (&idInst, (PFNCALLBACK) &DdeCallback,
                        CBF_FAIL_EXECUTES | CBF_FAIL_POKES |
                        CBF_SKIP_REGISTRATIONS | CBF_SKIP_UNREGISTRATIONS, 0))
          {
          MessageBox (hwnd, "Could not initialize server!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK) ;

          DestroyWindow (hwnd) ;
          return FALSE ;
          }

               // Set the timer

     SetTimer (hwnd, ID_TIMER, 5000, NULL) ;

               // Start things going

     SendMessage (hwnd, WM_USER_INITIATE, 0, 0L) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
```

```
          // Clean up

     DdeUninitialize (idInst) ;
     KillTimer (hwnd, ID_TIMER) ;

     return msg.wParam ;
     }

int GetStateNumber (UINT iFmt, HSZ hszItem)
     {
     char szItem[32] ;
     int  i ;

     if (iFmt != CF_TEXT)
          return -1 ;

     DdeQueryString (idInst, hszItem, szItem, sizeof (szItem), 0) ;

     for (i = 0 ; i < NUM_STATES ; i++)
          if (strcmp (szItem, pop[i].szState) == 0)
               break ;

     if (i >= NUM_STATES)
          return -1 ;

     return i ;
     }

HDDEDATA CALLBACK DdeCallback (UINT iType, UINT iFmt, HCONV hConv,
                               HSZ hsz1, HSZ hsz2, HDDEDATA hData,
                               DWORD dwData1, DWORD dwData2)
     {
     char szBuffer[32] ;
     int  i ;

     switch (iType)
          {
          case XTYP_CONNECT :            // hsz1 = topic
                                         // hsz2 = service

               DdeQueryString (idInst, hsz2, szBuffer, sizeof (szBuffer), 0) ;

               if (0 != strcmp (szBuffer, szAppName))
                    return FALSE ;

               DdeQueryString (idInst, hsz1, szBuffer, sizeof (szBuffer), 0) ;

               if (0 != strcmp (szBuffer, szTopic))
                    return FALSE ;

               return (HDDEDATA) TRUE ;

          case XTYP_ADVSTART :           // hsz1 = topic
                                         // hsz2 = item
```

```
                    // Check for matching format and data item

               if (-1 == (i = GetStateNumber (iFmt, hsz2)))
                    return FALSE ;

               pop[i].lPopLast = 0 ;
               PostMessage (hwnd, WM_TIMER, 0, 0L) ;

               return (HDDEDATA) TRUE ;

          case XTYP_REQUEST :
          case XTYP_ADVREQ :                 // hsz1 = topic
                                             // hsz2 = item

                    // Check for matching format and data item

               if (-1 == (i = GetStateNumber (iFmt, hsz2)))
                    return NULL ;

               wsprintf (szBuffer, "%ld\r\n", pop[i].lPop) ;

               return DdeCreateDataHandle (idInst, (unsigned char *) szBuffer,
                                           strlen (szBuffer) + 1,
                                           0, hsz2, CF_TEXT, 0) ;

          case XTYP_ADVSTOP :                // hsz1 = topic
                                             // hsz2 = item

                    // Check for matching format and data item

               if (-1 == (i = GetStateNumber (iFmt, hsz2)))
                    return FALSE ;

               return (HDDEDATA) TRUE ;
          }

     return NULL ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HSZ hszService, hszTopic ;
     HSZ        hszItem ;
     int        i ;

     switch (iMsg)
          {
          case WM_USER_INITIATE :
               InitPops () ;

               hszService = DdeCreateStringHandle (idInst, szAppName, 0) ;
               hszTopic   = DdeCreateStringHandle (idInst, szTopic,   0) ;

               DdeNameService (idInst, hszService, NULL, DNS_REGISTER) ;
```

```
               return 0 ;

          case WM_TIMER :
          case WM_TIMECHANGE :

                    // Calculate new current populations

               CalcPops () ;

               for (i = 0 ; i < NUM_STATES ; i++)
                    if (pop[i].lPop != pop[i].lPopLast)
                         {
                         hszItem = DdeCreateStringHandle (idInst,
                                                  pop[i].szState, 0) ;

                         DdePostAdvise (idInst, hszTopic, hszItem) ;

                         DdeFreeStringHandle (idInst, hszItem) ;

                         pop[i].lPopLast = pop[i].lPop ;
                         }

               return 0 ;

          case WM_QUERYOPEN :
               return 0 ;

          case WM_DESTROY :
               DdeNameService (idInst, hszService, NULL, DNS_UNREGISTER) ;
               DdeFreeStringHandle (idInst, hszService) ;
               DdeFreeStringHandle (idInst, hszTopic) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

DDEPOP2.RC

```
/*-------------------------
DDEPOP2.RC resource script
  -------------------------*/

DdePop2 ICON ddepop.ico
```

图 17-3　DDEPOP2程序

SHOWPOP2.MAK

```
#-----------------------
# SHOWPOP2.MAK make file
#-----------------------

showpop2.exe : showpop2.obj
     $(LINKER) $(GUIFLAGS) -OUT:showpop2.exe showpop2.obj $(GUILIBS)
```

```
showpop2.obj : showpop2.c showpop.h
     $(CC) $(CFLAGS) showpop2.c
```

SHOWPOP2.C

```
/*-----------------------------------------
   SHOWPOP2.C -- DDEML Client using DDEPOP2
                 (c) Charles Petzold, 1996
  -----------------------------------------*/

#include <windows.h>
#include <ddeml.h>
#include <stdlib.h>
#include <string.h>
#include "showpop.h"

#define WM_USER_INITIATE (WM_USER + 1)
#define DDE_TIMEOUT      3000

LRESULT  CALLBACK WndProc     (HWND, UINT, WPARAM, LPARAM) ;
HDDEDATA CALLBACK DdeCallback (UINT, UINT, HCONV, HSZ, HSZ,
                               HDDEDATA, DWORD, DWORD) ;

char  szAppName[] = "ShowPop2" ;
DWORD idInst ;
HCONV hConv ;
HWND  hwnd ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (hInstance, szAppName) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (hInstance, szAppName) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "DDEML Client - US Population",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
```

```
     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

               // Initialize for using DDEML


     if (DdeInitialize (&idInst, (PFNCALLBACK) &DdeCallback,
                        APPCLASS_STANDARD | APPCMD_CLIENTONLY, 0L))
          {
          MessageBox (hwnd, "Could not initialize client!",
                      szAppName, MB_ICONEXCLAMATION | MB_OK) ;

          DestroyWindow (hwnd) ;
          return FALSE ;
          }

               // Start things going

     SendMessage (hwnd, WM_USER_INITIATE, 0, 0L) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

               // Uninitialize DDEML

     DdeUninitialize (idInst) ;

     return msg.wParam ;
     }

HDDEDATA CALLBACK DdeCallback (UINT iType, UINT iFmt, HCONV hConv,
                               HSZ hsz1, HSZ hsz2, HDDEDATA hData,
                               DWORD dwData1, DWORD dwData2)
     {
     char szItem[10], szPopulation[16] ;
     int  i ;

     switch (iType)
          {
          case XTYP_ADVDATA :       // hsz1  = topic
                                    // hsz2  = item
                                    // hData = data

                    // Check for matching format and data item

               if (iFmt != CF_TEXT)
                    return DDE_FNOTPROCESSED ;

               DdeQueryString (idInst, hsz2, szItem, sizeof (szItem), 0) ;

               for (i = 0 ; i < NUM_STATES ; i++)
```

```
                    if (strcmp (szItem, pop[i].szAbb) == 0)
                         break ;

               if (i >= NUM_STATES)
                    return DDE_FNOTPROCESSED ;

                    // Store the data and invalidate the window

               DdeGetData (hData, (unsigned char *) szPopulation,
                           sizeof (szPopulation), 0) ;

               pop[i].lPop = atol (szPopulation) ;

               InvalidateRect (hwnd, NULL, FALSE) ;

               return (HDDEDATA) DDE_FACK ;

          case XTYP_DISCONNECT :
               hConv = NULL ;

               MessageBox (hwnd, "The server has disconnected.",
                           szAppName, MB_ICONASTERISK | MB_OK) ;

               return NULL ;
          }

     return NULL ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static char  szService[] = "DdePop2",
                  szTopic[]   = "US_Population" ;
     static long  cxChar, cyChar ;
     char         szBuffer[24] ;
     HDC          hdc ;
     HSZ          hszService, hszTopic, hszItem ;
     PAINTSTRUCT  ps ;
     int          i ;
     long         x, y ;
     TEXTMETRIC   tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
               ReleaseDC (hwnd, hdc) ;

               return 0 ;

          case WM_USER_INITIATE :
```

```
               // Try connecting

          hszService = DdeCreateStringHandle (idInst, szService, 0) ;
          hszTopic   = DdeCreateStringHandle (idInst, szTopic,   0) ;

          hConv = DdeConnect (idInst, hszService, hszTopic, NULL) ;

               // If that doesn't work, load server

          if (hConv == NULL)
               {
               WinExec (szService, SW_SHOWMINNOACTIVE) ;

               hConv = DdeConnect (idInst, hszService, hszTopic, NULL) ;
               }

               // Free the string handles

          DdeFreeStringHandle (idInst, hszService) ;
          DdeFreeStringHandle (idInst, hszTopic) ;

               // If still not connected, display message box

          if (hConv == NULL)
               {
               MessageBox (hwnd, "Cannot connect with DDEPOP2.EXE!",
                           szAppName, MB_ICONEXCLAMATION | MB_OK) ;

               return 0 ;
               }

               // Request notification

          for (i = 0 ; i < NUM_STATES ; i++)
               {
               hszItem = DdeCreateStringHandle (idInst, pop[i].szAbb, 0) ;

               DdeClientTransaction (NULL, 0, hConv, hszItem, CF_TEXT,
                                     XTYP_ADVSTART | XTYPF_ACKREQ,
                                     DDE_TIMEOUT, NULL) ;

               DdeFreeStringHandle (idInst, hszItem) ;
               }

          if (i < NUM_STATES)
               {
               MessageBox (hwnd, "Failure on WM_DDE_ADVISE!",
                           szAppName, MB_ICONEXCLAMATION | MB_OK) ;
               }

          return 0 ;

     case WM_PAINT :
          hdc = BeginPaint (hwnd, &ps) ;
```

```
          for (i = 0 ; i < NUM_STATES ; i++)
               {
               if (i < (NUM_STATES + 1) / 2)
                    {
                    x = cxChar ;
                    y = i * cyChar ;
                    }
               else
                    {
                    x = 44 * cxChar ;
                    y = (i - (NUM_STATES + 1) / 2) * cyChar ;
                    }

               TextOut (hdc, x, y, szBuffer,
                        wsprintf (szBuffer, "%-20s",
                                  (PSTR) pop[i].szState)) ;

               x += 36 * cxChar ;

               SetTextAlign (hdc, TA_RIGHT * TA_TOP) ;

               TextOut (hdc, x, y, szBuffer,
                        wsprintf (szBuffer, "%10ld", pop[i].lPop)) ;

               SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
               }

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_CLOSE :
          if (hConv == NULL)
               break ;

                    // Stop the advises

          for (i = 0 ; i < NUM_STATES ; i++)
               {
               hszItem = DdeCreateStringHandle (idInst, pop[i].szAbb, 0) ;

               DdeClientTransaction (NULL, 0, hConv, hszItem, CF_TEXT,
                                     XTYP_ADVSTOP, DDE_TIMEOUT, NULL) ;

               DdeFreeStringHandle (idInst, hszItem) ;
               }

                    // Disconnect the conversation

          DdeDisconnect (hConv) ;

          break ;              // for default processing

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
```

```
	}
	return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

图 17-4　DDEPOP2程序

在WinMain中，SHOWPOP2将其回调函数(叫做DdeCallback)的地址传送给DdeInitialize。DDEML实例句柄保存在全局变量idInst中，它必须作为其它所有DDEML函数调用的第一个参数。

然后SHOWPOP2给它自己发送一条WM_USER_INITLATE消息；其它初始化是在WndProc窗口过程中完成的。该程序两次调用DdeCreateStringHandle来为服务器的服务名(“DdePop2”)和主题名(“US_Population”)创建串句柄。SHOWPOP2把这些串句柄传给DdeConnect来试图发动与DDEPOP2服务器的对话。如SHOWPOP1中所示，如果不能发动对话，程序就调用WinExec来加载DDEPOP2并重试一次。随后，可以调用DdeFreeStringHandle释放串句柄。

在DDEPOP2服务器启动后，它在其WinMain函数中调用DdeInitializde，同时调用SetTimer为更新人口总数设置一个5秒间隔的定时器，然后DDEPOP2向它的窗口过程发送一条WM_USER_INITIATE消息。处理这条消息时，DDEPOP2初始化人口数据并为其服务名和主题名创建两个串句柄。然后WndProc调用DdeNameService注册它的服务名，这将阻止回调函数接受任何没有明确定义为服务的连接请求。

当SHOWPOP2调用DdeConnect时，DDE库管理程序用XTYP_CONNECT事务类型调用DDEPOP2中的回调函数，回调函数的hsz1和hsz2参数是传送给DdeConnect函数的主题名和服务名的串句柄。DDEPOP2中的DdeCallBack函数使用DdeQueryString来检查主题名和服务名是否与串“US_Population”及“DdePop2”匹配。如果匹配，DdeCallBack将返回TRUE；否则返回FALSE。回调函数不需要释放传递给它的串句柄。

当服务器的DdeCallBack函数处理完XTYP_CONNECT事务后返回TRUE时，DDE对话就开始了。由客户(SHOWPOP2)调用的DdeConnect函数返回一个对话句柄，它的数据类型是HCONV。这个句柄也在XTYP_CONNECT_CONFIRM事务中被传给服务器的回调函数。如果有必要，服务器将保存这个对话句柄，而DDEPOP2却不能保存它。

最后剩下SHOWPOP2，它已通过调用DdeConnect创建了同DDEPOP2的连接，并释放了该函数所要求的服务名和主题名的串句柄。随后，SHOWPOP2开始使用热链接，来统计50个州、哥伦比亚特区和全美国的人口数。为此，SHOWPOP2进入一个循环并用每个双字符州代码调用DdeCreateStringHandle。这些值连同对话句柄、剪贴板格式类型和XTYP_ADVSTART标志一起传送给DdeClientTransaction。这是一个客户用来从服务器得到数据的基本函数。然后调用DdeFreeStringHandle释放每个串句柄。

当参加对话的客户用XTYP_ADVSTART参数调用DdeClientTransaction时，DDE管理库用XTYP_ADVSTART事务类型调用服务器的回调函数。DDEPOP2只是简单地检查剪贴板格式是否为CF_TEXT，项名是否为有效的双字符州代码(在DDEPOP2的GetStateNumber函数中完成)。如果项名有效，DDE就把这个州的人口数设置为0，并把WM_TIMER消息邮递给它的窗口过程。

这里正是DDEML的关键所在。DDEPOP2服务器“同意”对话，但它没有保存对话

句柄。 DDEPOP2 回调函数也收到了特定州的 XTYP_ADVSTART 事务类型，但它同样没有保存关于哪个客户在哪次对话中需要哪个州的人口数据的信息。

DDEPOP2 没有保存这个信息，是因为 DDEML 保存了它。现在让我们看看在 WndProc 中处理 WM_TIMER(和 WM_TIMECHANGE)期间 DDEPOP2 更新它的人口信息时会发生什么情况。它调用 CalcPops 计算新的人口总数，然后检查哪个州的人口数发生了变化。如果某个州的人口数发生了变化， DDEPOP2 就为这个州的双字符州代码创建串句柄，然后用主题名和项名调用 DdePostAdvise 。

在每次对话中，当客户将主题名和项名传给 DdePostAdvise 并请求热链接时，DdePostAdvise 就用 XTYP_ADVREQ 事务类型调用 DDEPOP2 自己的回调函数，回调函数的 hsz1 和 hsz2 参数指定主题名和项名。回调函数所需做的所有工作是返回新数据的数据句柄。为此, DdeCallBack 格式化人口数据串，并将串缓冲区传送给 DdeCreateDataHandle 。这个函数返回一个 HDDEDATA 类型的数据句柄，回调函数直接返回该句柄。从回调函数返回的数据句柄不需要被显式地释放。

当 DDEPOP2 从它的回调函数返回一个数据句柄作为对 XTYP_ADVREQ 事务的响应时， DDE 管理库以 XTYP_ADVDATA 事务类型调用 SHOWPOP2 的回调函数。SHOWPOP2 使用 DdeGetData 函数得到人口数据并保存此数据。然后， SHOWPOP2 使其窗口无效, 并更新显示。

DDEML 的使用大致就是这样的。当 SHOWPOP2 结束时，它使用 XTYP_ADVSTOP 事务类型为所有州调用 DdeClientTransaction 。 DDEPOP2 在它的回调函数中接收这个事务。然后， SHOWPOP2 调用 DdeDisconnect，并在退出消息循环之后调用 DdeUninitialize 。DDE 程序在此函数之前未完成的任何结束处理均由 DdeUninitialize 自动完成。

第 18 章　多文档界面

多文档界面(MDI)是 Microsoft Windows 文档处理应用程序的一种规范，该规范描述了窗口结构和允许用户在单个应用程序中使用多个文档的用户界面（如字处理程序中的文本文档和电子表格程序中的电子表格）。简单地说，就像 Windows 在一个屏幕上维护多个应用程序窗口一样， MDI 应用程序在一个客户区内维护多个文档窗口。 Windows 中的第一个 MDI 应用程序是 Windows 下的 Microsoft Excel 的第一个版本。 Microsoft Word for Windows 和 Microsoft Access 都是 MDI 应用。

尽管 MDI 规范随着 Windows 2 的推出已经很普及，但在那时， MDI 应用程序写起来很困难，并且需要一些非常复杂的程序设计工作。从 Windows 3 起，其中许多工作就都已经为你做好了。 Windows 95 只是在专门用于简化 MDI 应用的那组函数、数据结构和消息集合中增加了一个新函数和一条新消息。

18.1　MDI 的组成

MDI 程序的主应用程序窗口是很普通的：它有一个标题条、一个菜单、一个缩放边框、一个系统菜单图标和最大化/最小化图标。客户区经常被称为“工作空间”，它不直接用于显示程序输出。这个工作空间包括零个或多个子窗口，每个窗口都显示一个文档。

这些子窗口看起来与通常的应用程序窗口很相似。它们有一个标题条、一个缩放边框、一个系统菜单图标和最大化/最小化的图标，可能还包括滚动条。但是文档窗口没有菜单，主应用程序窗口上的菜单适用于文档窗口。

在任何时候都只能有一个文档窗口是活动的（加亮标题条来表示），它出现在其它所有文档窗口之前。所有文档窗口都由工作空间区域加以剪裁，而不会出现在应用程序窗口之外。

初看起来，对 Windows 程序员来说， MDI 似乎是相当简单。需要程序员做的工作好像就是为每个文档创建一个 WM_CHILD 窗口，并使程序的主应用程序窗口成为文档窗口的父窗口。但对诸如 Microsoft Excel 这样的 MDI 应用程序稍加研究，就会发现一些导致编码困难的复杂问题。例如：

- MDI 文档窗口可以最小化。它的图标出现在工作空间的底部(一般来说， MDI 应用程序可以将不同的图标分别用于主应用程序窗口和每一类文档应用)。
- MDI 文档窗口可以最大化。在这种情况下，文档窗口的标题条(一般用来显示窗口中文档的文件名)消失，文件名出现在应用程序窗口标题条的应用程序名之后，文档窗口的系统菜单图标成为应用程序窗口的顶层菜单中的第一项。恢复文档窗口

大小的图标变成顶层菜单中的最后一项，且出现在最右边。

- 用以关闭文档窗口的系统键盘加速键与关闭主窗口的系统键盘加速键一样，只是 Ctrl 键代替了 Alt 键。这也就是说， Alt+F4 用于关闭应用程序窗口，而 Ctrl+F4 用于关闭文档窗口。此外， Ctrl+F6 可以在活动 MDI 应用程序的子文档窗口之间切换。与平时一样， Alt+空格键激活主窗口的系统菜单， Alt+-(减号)激活活动子文档窗口的系统菜单。
- 当使用光标键在菜单项间移动时，控制通常从系统菜单转到菜单条中的第一项。在 MDI 应用程序中，控制是从应用程序系统菜单转到活动文档系统菜单，然后再转到菜单条中的第一项。
- 如果应用程序能够支持若干种类型的子窗口(如 Microsoft Excel 中的工作表和图表文档),那么菜单应能反映出与这种类型的文档有关的操作。这就要求当不同的文本窗口变成活动窗口时，程序能更换菜单。此外，当没有文档窗口存在时，菜单应该被缩减到只剩下与打开新文档有关的操作。
- 顶层菜单上有一个叫做“窗口(Window)”的菜单项。按照习惯，这是顶层菜单上 Help 之前的那一项，即倒数第二项。“窗口”子菜单上通常包含在工作空间内安排文档窗口的选项。文档窗口可以从左上方开始“平铺”或“层叠”。在前一种方式下，可以完整地看到每一个文档窗口。这个子菜单同时也包含所有文档窗口的列表。从中选择一个文档窗口，就可以把此文档窗口移到前景上。

Windows 95 支持 MDI 的所有这些方面。当然，需要你做一些工作（如下面的示例程序所示），但是，这远不是要你编写程序来直接支持所有这些功能。

18.2 WINDOWS 95 和 MDI

探讨 Windows 95 的 MDI 支持时需要引入一些新术语。主应用程序窗口称为“框架窗口”，就像传统的 Windows 程序一样，它是 WS_OVERLAPPEDWINDOW 风格的窗口。

MDI 应用程序还根据预定义的窗口类“ MDICLIENT ”创建“客户窗口”，这一客户窗口是用这种窗口类和 WS_CHILD 风格调用 CreateWindow 来创建的。这一调用的最后一个参数是指向一个 CLIENTCREATESTRUCT 类型的结构的指针。这个客户窗口覆盖框架窗口的客户区，并提供许多 MDI 支持。此客户窗口的颜色是系统颜色 COLOR_APPWORDSPACE 。

文档窗口被称为“子窗口”。通过初始化一个 MDICREATESTRUCT 类型的结构，以一个指向此结构的指针为参数将消息 WM_MDICREATE 发送给客户窗口，就可以创建这些文档窗口。

文档窗口是客户窗口的子窗口，而客户窗口又是框架窗口的子窗口。父-子窗口分层结构如图 18-1 所示。

你需要框架窗口的窗口类（及窗口过程）和一个由应用程序支持的每类子窗口的窗口类（及窗口过程）。由于已经预先注册了窗口类，所以不需要客户窗口的窗口过程。

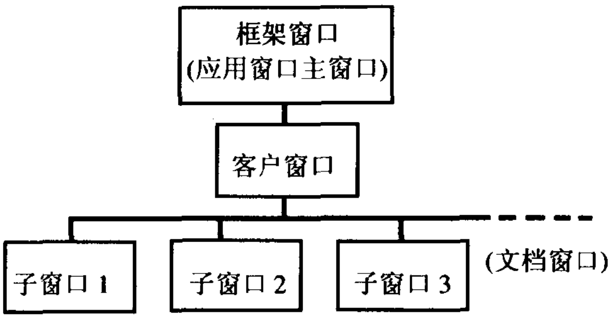


图 18-1 Windows MDI应用的父-子层次图

Windows95 的 MDI 支持包括一个窗口类、五个函数、两个数据结构和 12 条消息。前面已经提到了 MDI 窗口类，即 MDICLIENT，以及数据结构 CLIENTCREATESTRUCT 和 MDICREATESTRUCT 。在 MDI 应用程序中，这五个函数中的两个用于取代 DefWindowsProc ：不再将 DefWindowProc 调用用于所有未处理的消息，而是由框架窗口过程调用 DefFrameProc, 子窗口过程调用 DefMDIChildProc 。另一个新函数 TranslateMDISysAccel 与第 9 章中讨论的 TranslateAccelerator 的使用方式相同。Windows 3 增加了函数 ArrangeIconicWindows，但有一条专用的 MDI 消息使得此函数对 MDI 程序来说不再必要。

如果 MDI 子窗口要在你的程序中进行任何长时间的操作，那么应该考虑将这些操作放在独立的线程中运行。这允许用户从一个忙碌的子窗口切换出来，继续在另一个窗口工作，而让第一个窗口在后台完成它的任务。 Windows 95 提供了专门用于这个目的的新函数 CreateMDIWindow 。线程调用 CreateMDIWindow 来创建一个 MDI 子窗口；此后，窗口操作完全发生在这个线程的环境中。一个单线程程序不需要用 CreateMDIWindow 来创建子窗口，因为 WM_MDICREATE 消息可以完成同样的工作。

在下面给出的单线程示例程序中，我将演示 12 条 MDI 消息中的 9 条(其它消息一般不用)，这些消息的前缀是 WM_MDI 。框架窗口向客户窗口发送其中某条消息，以便在子窗口上完成一项操作或者获取有关子窗口的信息（例如，框架窗口发送一条 WM_MDICREATE 消息给客户窗口，以创建子窗口)。消息 WM_MDIACTIVATE 有点特别：框架窗口可以发送这条消息给客户窗口来激活一个子窗口，而客户窗口也把这条消息发送给将被激活或者失活的子窗口，以便通知它们这一变化。

18.3 示例程序

图 18-2 给出的 MDIDEMO 程序说明了编写 MDI 应用程序的基本方法。

MDIDEMO.MAK

```
#-----------------------
# MDIDEMO.MAK make file
#-----------------------

mdidemo.exe : mdidemo.obj mdidemo.res
```

```
    $(LINKER) $(GUIFLAGS) -OUT:mdidemo.exe mdidemo.obj mdidemo.res $(GUILIBS)

mdidemo.obj : mdidemo.c mdidemo.h
    $(CC) $(CFLAGS) mdidemo.c

mdidemo.res : mdidemo.rc mdidemo.h
    $(RC) $(RCVARS) mdidemo.rc
```

MDIDEMO.C

```
/*--------------------------------------------------------
   MDIDEMO.C -- Multiple Document Interface Demonstration
                (c) Charles Petzold, 1996
  --------------------------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include "mdidemo.h"

LRESULT CALLBACK FrameWndProc  (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK CloseEnumProc (HWND, LPARAM) ;
LRESULT CALLBACK HelloWndProc  (HWND, UINT, WPARAM, LPARAM) ;
LRESULT CALLBACK RectWndProc   (HWND, UINT, WPARAM, LPARAM) ;

          // structure for storing data unique to each Hello child window

typedef struct tagHELLODATA
     {
     UINT     iColor ;
     COLORREF clrText ;
     }
     HELLODATA, *LPHELLODATA ;

          // structure for storing data unique to each Rect child window

typedef struct tagRECTDATA
     {
     short cxClient ;
     short cyClient ;
     }
     RECTDATA, *LPRECTDATA ;

          // global variables

char      szFrameClass[] = "MdiFrame" ;
char      szHelloClass[] = "MdiHelloChild" ;
char      szRectClass[]  = "MdiRectChild" ;
HINSTANCE hInst ;
HMENU     hMenuInit, hMenuHello, hMenuRect ;
HMENU     hMenuInitWindow, hMenuHelloWindow, hMenuRectWindow ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
                  PSTR szCmdLine, int iCmdShow)
{
HACCEL      hAccel ;
HWND        hwndFrame, hwndClient ;
MSG         msg ;
WNDCLASSEX  wndclass ;

hInst = hInstance ;

if (!hPrevInstance)
     {
                  // Register the frame window class

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = FrameWndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) (COLOR_APPWORKSPACE + 1) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szFrameClass ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

                  // Register the Hello child window class

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = HelloWndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = sizeof (HANDLE) ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szHelloClass ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

                  // Register the Rect child window class

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = RectWndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = sizeof (HANDLE) ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
```

```
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szRectClass ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;
     }
          // Obtain handles to three possible menus & submenus

     hMenuInit  = LoadMenu (hInst, "MdiMenuInit") ;
     hMenuHello = LoadMenu (hInst, "MdiMenuHello") ;
     hMenuRect  = LoadMenu (hInst, "MdiMenuRect") ;

     hMenuInitWindow  = GetSubMenu (hMenuInit,   INIT_MENU_POS) ;
     hMenuHelloWindow = GetSubMenu (hMenuHello, HELLO_MENU_POS) ;
     hMenuRectWindow  = GetSubMenu (hMenuRect,   RECT_MENU_POS) ;

          // Load accelerator table

     hAccel = LoadAccelerators (hInst, "MdiAccel") ;

          // Create the frame window

     hwndFrame = CreateWindow (szFrameClass, "MDI Demonstration",
                               WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               CW_USEDEFAULT, CW_USEDEFAULT,
                               NULL, hMenuInit, hInstance, NULL) ;

     hwndClient = GetWindow (hwndFrame, GW_CHILD) ;

     ShowWindow (hwndFrame, iCmdShow) ;
     UpdateWindow (hwndFrame) ;

          // Enter the modified message loop

     while (GetMessage (&msg, NULL, 0, 0))
          {
          if (!TranslateMDISysAccel (hwndClient, &msg) &&
              !TranslateAccelerator (hwndFrame, hAccel, &msg))
               {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
               }
          }
          // Clean up by deleting unattached menus

     DestroyMenu (hMenuHello) ;
     DestroyMenu (hMenuRect) ;

     return msg.wParam ;
     }

LRESULT CALLBACK FrameWndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                     LPARAM lParam)
```

```
{
static HWND          hwndClient ;
CLIENTCREATESTRUCT   clientcreate ;
HWND                 hwndChild ;
MDICREATESTRUCT      mdicreate ;

switch (iMsg)
     {
     case WM_CREATE :          // Create the client window

          clientcreate.hWindowMenu  = hMenuInitWindow ;
          clientcreate.idFirstChild = IDM_FIRSTCHILD ;

          hwndClient = CreateWindow ("MDICLIENT", NULL,
                         WS_CHILD | WS_CLIPCHILDREN | WS_VISIBLE,
                         0, 0, 0, 0, hwnd, (HMENU) 1, hInst,
                         (LPSTR) &clientcreate) ;
          return 0 ;

     case WM_COMMAND :
          switch (wParam)
               {
               case IDM_NEWHELLO :       // Create a Hello child window

                    mdicreate.szClass = szHelloClass ;
                    mdicreate.szTitle = "Hello" ;
                    mdicreate.hOwner  = hInst ;
                    mdicreate.x       = CW_USEDEFAULT ;
                    mdicreate.y       = CW_USEDEFAULT ;
                    mdicreate.cx      = CW_USEDEFAULT ;
                    mdicreate.cy      = CW_USEDEFAULT ;
                    mdicreate.style   = 0 ;
                    mdicreate.lParam  = 0 ;

                    hwndChild = (HWND) SendMessage (hwndClient,
                              WM_MDICREATE, 0,
                                (LPARAM) (LPMDICREATESTRUCT) &mdicreate) ;
                    return 0 ;

               case IDM_NEWRECT :        // Create a Rect child window

                    mdicreate.szClass = szRectClass ;
                    mdicreate.szTitle = "Rectangles" ;
                    mdicreate.hOwner  = hInst ;
                    mdicreate.x       = CW_USEDEFAULT ;
                    mdicreate.y       = CW_USEDEFAULT ;
                    mdicreate.cx      = CW_USEDEFAULT ;
                    mdicreate.cy      = CW_USEDEFAULT ;
                    mdicreate.style   = 0 ;
                    mdicreate.lParam  = 0 ;

                    hwndChild = (HWND) SendMessage (hwndClient,
                              WM_MDICREATE, 0,
                                (LPARAM) (LPMDICREATESTRUCT) &mdicreate) ;
                    return 0 ;
```

```
                    case IDM_CLOSE :          // Close the active window

                         hwndChild = (HWND) SendMessage (hwndClient,
                                                  WM_MDIGETACTIVE, 0, 0) ;

                         if (SendMessage (hwndChild, WM_QUERYENDSESSION, 0, 0))
                              SendMessage (hwndClient, WM_MDIDESTROY,
                                           (WPARAM) hwndChild, 0) ;
                         return 0 ;

                    case IDM_EXIT :           // Exit the program

                         SendMessage (hwnd, WM_CLOSE, 0, 0) ;
                         return 0 ;

                                   // messages for arranging windows
                    case IDM_TILE :
                         SendMessage (hwndClient, WM_MDITILE, 0, 0) ;
                         return 0 ;

                    case IDM_CASCADE :
                         SendMessage (hwndClient, WM_MDICASCADE, 0, 0) ;
                         return 0 ;

                    case IDM_ARRANGE :
                         SendMessage (hwndClient, WM_MDIICONARRANGE, 0, 0) ;
                         return 0 ;

                    case IDM_CLOSEALL :       // Attempt to close all children

                         EnumChildWindows (hwndClient, &CloseEnumProc, 0) ;
                         return 0 ;

                    default:             // Pass to active child...

                         hwndChild = (HWND) SendMessage (hwndClient,
                                                  WM_MDIGETACTIVE, 0, 0) ;

                        if (IsWindow (hwndChild))
                             SendMessage (hwndChild, WM_COMMAND,
                                          wParam, lParam) ;

                        break ;        // ...and then to DefFrameProc
                    }
               break ;

          case WM_QUERYENDSESSION :
          case WM_CLOSE :                      // Attempt to close all children

               SendMessage (hwnd, WM_COMMAND, IDM_CLOSEALL, 0) ;

               if (NULL != GetWindow (hwndClient, GW_CHILD))
                    return 0 ;
```

```
               break ;    // I.e., call DefFrameProc

          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
               // Pass unprocessed messages to DefFrameProc (not DefWindowProc)

     return DefFrameProc (hwnd, hwndClient, iMsg, wParam, lParam) ;
     }

BOOL CALLBACK CloseEnumProc (HWND hwnd, LPARAM lParam)
     {
     if (GetWindow (hwnd, GW_OWNER))          // Check for icon title
          return 1 ;

     SendMessage (GetParent (hwnd), WM_MDIRESTORE, (WPARAM) hwnd, 0) ;

     if (!SendMessage (hwnd, WM_QUERYENDSESSION, 0, 0))
          return 1 ;

     SendMessage (GetParent (hwnd), WM_MDIDESTROY, (WPARAM) hwnd, 0) ;
          return 1 ;
     }

LRESULT CALLBACK HelloWndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                     LPARAM lParam)
     {
     static COLORREF clrTextArray[] = { RGB (0,    0, 0), RGB (255, 0,   0),
                                        RGB (0, 255, 0), RGB ( 0, 0, 255),
                                        RGB (255, 255, 255) } ;
     static HWND     hwndClient, hwndFrame ;
     HDC             hdc ;
     HMENU           hMenu ;
     LPHELLODATA     lpHelloData ;
     PAINTSTRUCT     ps ;
     RECT            rect ;

     switch (iMsg)
          {
          case WM_CREATE :
                         // Allocate memory for window private data

               lpHelloData = (LPHELLODATA) HeapAlloc (GetProcessHeap (),
                                                      HEAP_ZERO_MEMORY,
                                                      sizeof (HELLODATA)) ;
               lpHelloData->iColor  = IDM_BLACK ;
               lpHelloData->clrText = RGB (0, 0, 0) ;
               SetWindowLong (hwnd, 0, (long) lpHelloData) ;

                         // Save some window handles

               hwndClient = GetParent (hwnd) ;
               hwndFrame  = GetParent (hwndClient) ;
               return 0 ;
```

```
case WM_COMMAND :
     switch (wParam)
          {
          case IDM_BLACK :
          case IDM_RED :
          case IDM_GREEN :
          case IDM_BLUE :
          case IDM_WHITE :
                         // Change the text color

               lpHelloData = (LPHELLODATA) GetWindowLong (hwnd, 0) ;

               hMenu = GetMenu (hwndFrame) ;

               CheckMenuItem (hMenu, lpHelloData->iColor,
                                     MF_UNCHECKED) ;
               lpHelloData->iColor = wParam ;
               CheckMenuItem (hMenu, lpHelloData->iColor,
                                     MF_CHECKED) ;

               lpHelloData->clrText =
                     clrTextArray[wParam - IDM_BLACK] ;

               InvalidateRect (hwnd, NULL, FALSE) ;
          }
     return 0 ;

case WM_PAINT :
               // Paint the window

     hdc = BeginPaint (hwnd, &ps) ;

     lpHelloData = (LPHELLODATA) GetWindowLong (hwnd, 0) ;
     SetTextColor (hdc, lpHelloData->clrText) ;

     GetClientRect (hwnd, &rect) ;

     DrawText (hdc, "Hello, World!", -1, &rect,
               DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

     EndPaint (hwnd, &ps) ;
     return 0 ;

case WM_MDIACTIVATE :

               // Set the Hello menu if gaining focus

     if (lParam == (LPARAM) hwnd)
          SendMessage (hwndClient, WM_MDISETMENU,
                      (WPARAM) hMenuHello, (LPARAM) hMenuHelloWindow) ;

               // Check or uncheck menu item

     lpHelloData = (LPHELLODATA) GetWindowLong (hwnd, 0) ;
```

```
               CheckMenuItem (hMenuHello, lpHelloData->iColor,
                         (lParam == (LPARAM) hwnd) ? MF_CHECKED : MF_UNCHECKED) ;

                         // Set the Init menu if losing focus

               if (lParam != (LPARAM) hwnd)
                    SendMessage (hwndClient, WM_MDISETMENU, (WPARAM) hMenuInit,
                                 (LPARAM) hMenuInitWindow) ;

               DrawMenuBar (hwndFrame) ;
               return 0 ;

          case WM_QUERYENDSESSION :
          case WM_CLOSE :
               if (IDOK != MessageBox (hwnd, "OK to close window?", "Hello",
                                       MB_ICONQUESTION | MB_OKCANCEL))
                    return 0 ;

               break ;   // I.e., call DefMDIChildProc

          case WM_DESTROY :
               lpHelloData = (LPHELLODATA) GetWindowLong (hwnd, 0) ;
               HeapFree (GetProcessHeap (), 0, lpHelloData) ;
               return 0 ;
          }
               // Pass unprocessed message to DefMDIChildProc

     return DefMDIChildProc (hwnd, iMsg, wParam, lParam) ;
     }

LRESULT CALLBACK RectWndProc (HWND hwnd, UINT iMsg, WPARAM wParam,
                                                    LPARAM lParam)
     {
     static HWND  hwndClient, hwndFrame ;
     HBRUSH       hBrush ;
     HDC          hdc ;
     LPRECTDATA   lpRectData ;
     PAINTSTRUCT  ps ;
     int          xLeft, xRight, yTop, yBottom ;
     short        nRed, nGreen, nBlue ;

     switch (iMsg)
          {
          case WM_CREATE :
                         // Allocate memory for window private data

               lpRectData = (LPRECTDATA) HeapAlloc (GetProcessHeap (),
                                                    HEAP_ZERO_MEMORY,
                                                    sizeof (RECTDATA)) ;

               SetWindowLong (hwnd, 0, (long) lpRectData) ;

                         // Start the timer going

               SetTimer (hwnd, 1, 250, NULL) ;
```

```
                    // Save some window handles

          hwndClient = GetParent (hwnd) ;
          hwndFrame  = GetParent (hwndClient) ;
          return 0 ;

     case WM_SIZE :             // If not minimized, save the window size

          if (wParam != SIZE_MINIMIZED)
               {
               lpRectData = (LPRECTDATA) GetWindowLong (hwnd, 0) ;

               lpRectData->cxClient = LOWORD (lParam) ;
               lpRectData->cyClient = HIWORD (lParam) ;
               }

          break ;        // WM_SIZE must be processed by DefMDIChildProc

     case WM_TIMER :            // Display a random rectangle

          lpRectData = (LPRECTDATA) GetWindowLong (hwnd, 0) ;

          xLeft   = rand () % lpRectData->cxClient ;
          xRight  = rand () % lpRectData->cxClient ;
          yTop    = rand () % lpRectData->cyClient ;
          yBottom = rand () % lpRectData->cyClient ;
          nRed    = rand () & 255 ;
          nGreen  = rand () & 255 ;
          nBlue   = rand () & 255 ;

          hdc = GetDC (hwnd) ;
          hBrush = CreateSolidBrush (RGB (nRed, nGreen, nBlue)) ;
          SelectObject (hdc, hBrush) ;

          Rectangle (hdc, min (xLeft, xRight), min (yTop, yBottom),
                          max (xLeft, xRight), max (yTop, yBottom)) ;

          ReleaseDC (hwnd, hdc) ;
          DeleteObject (hBrush) ;
          return 0 ;

     case WM_PAINT :            // Clear the window

          InvalidateRect (hwnd, NULL, TRUE) ;
          hdc = BeginPaint (hwnd, &ps) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_MDIACTIVATE :      // Set the appropriate menu
          if (lParam == (LPARAM) hwnd)
               SendMessage (hwndClient, WM_MDISETMENU, (WPARAM) hMenuRect,
                            (LPARAM) hMenuRectWindow) ;
          else
               SendMessage (hwndClient, WM_MDISETMENU, (WPARAM) hMenuInit,
```

```
                                        (LPARAM) hMenuInitWindow) ;

                    DrawMenuBar (hwndFrame) ;
                    return 0 ;

               case WM_DESTROY :
                    lpRectData = (LPRECTDATA) GetWindowLong (hwnd, 0) ;
                    HeapFree (GetProcessHeap (), 0, lpRectData) ;
                    KillTimer (hwnd, 1) ;
                    return 0 ;
               }
                    // Pass unprocessed message to DefMDIChildProc

     return DefMDIChildProc (hwnd, iMsg, wParam, lParam) ;
     }
```

MDIDEMO.RC

```
/*----------------------------
   MDIDEMO.RC resource script
  ----------------------------*/

#include <windows.h>
#include "mdidemo.h"

MdiMenuInit MENU
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",             IDM_NEWHELLO
          MENUITEM "New &Rectangles",        IDM_NEWRECT
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     }

MdiMenuHello MENU
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",             IDM_NEWHELLO
          MENUITEM "New &Rectangles",        IDM_NEWRECT
          MENUITEM "&Close",                 IDM_CLOSE
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                  IDM_EXIT
          }
     POPUP "&Color"
          {
          MENUITEM "&Black",                 IDM_BLACK
          MENUITEM "&Red",                   IDM_RED
          MENUITEM "&Green",                 IDM_GREEN
          MENUITEM "B&lue",                  IDM_BLUE
          MENUITEM "&White",                 IDM_WHITE
          }
     POPUP "&Window"
```

```
          {
          MENUITEM "&Cascade\tShift+F5",      IDM_CASCADE
          MENUITEM "&Tile\tShift+F4",         IDM_TILE
          MENUITEM "Arrange &Icons",          IDM_ARRANGE
          MENUITEM "Close &All",              IDM_CLOSEALL
          }
     }
MdiMenuRect MENU
     {
     POPUP "&File"
          {
          MENUITEM "New &Hello",              IDM_NEWHELLO
          MENUITEM "New &Rectangles",         IDM_NEWRECT
          MENUITEM "&Close",                  IDM_CLOSE
          MENUITEM SEPARATOR
          MENUITEM "E&xit",                   IDM_EXIT
          }
     POPUP "&Window"
          {
          MENUITEM "&Cascade\tShift+F5",      IDM_CASCADE
          MENUITEM "&Tile\tShift+F4",         IDM_TILE
          MENUITEM "Arrange &Icons",          IDM_ARRANGE
          MENUITEM "Close &All",              IDM_CLOSEALL
          }
     }

MdiAccel ACCELERATORS
     {
     VK_F5, IDM_CASCADE, VIRTKEY, SHIFT
     VK_F4, IDM_TILE,    VIRTKEY, SHIFT
     }
```

MDIDEMO.H

```
/*-----------------------
   MDIDEMO.H header file
  -----------------------*/

#define EXPORT              __declspec (dllexport)

#define INIT_MENU_POS       0
#define HELLO_MENU_POS      2
#define RECT_MENU_POS       1

#define IDM_NEWHELLO       10
#define IDM_NEWRECT        11
#define IDM_CLOSE          12
#define IDM_EXIT           13

#define IDM_BLACK          20
#define IDM_RED            21
#define IDM_GREEN          22
#define IDM_BLUE           23
#define IDM_WHITE          24
```

```
#define IDM_TILE        30
#define IDM_CASCADE     31
#define IDM_ARRANGE     32
#define IDM_CLOSEALL    33

#define IDM_FIRSTCHILD 100
```

图 18-2 MDIDEMO程序

MDIDEMO 支持两种类型的非常简单的文档窗口：第一种窗口在它的客户区中央显示"Hello, World!"，另一种窗口显示一系列随机矩形（在源代码清单和标识符名中，它们分别叫做 Hello 文档和 Rect 文档）。这两类文档窗口的菜单不同，显示" Hello, World!"的文档窗口有一个允许用户修改文本颜色的菜单。

18.3.1 三个菜单

现在让我们先看看 MDIDEMO.RC 资源描述文件，它定义了程序所使用的三个菜单模板。

当文档窗口不存在时，程序显示 MdiMenuInit 菜单，这个菜单只允许用户创建新文档或退出程序。

MdiMenuHello 菜单与显示" Hello, World!"的文档窗口相关联。File 子菜单允许用户打开任何一类新文档、关闭活动文档或退出程序。 Color 子菜单允许用户设置文本颜色。Window 子菜单包括以平铺或者重叠的方式安排文档窗口、安排文档图标或关闭所有窗口等选项，这个子菜单也列出所有文档窗口的名称。

MdiMenuRect 菜单与随机矩形文档相关联。除了不包含 Color 子菜单外，它与 MdiMenuHello 菜单一样。

MDIDEMO.H 头文件定义了所有菜单标识符和三个常量:

```
#define  INIT_MENU_POS  0
#define  HELLO_MENU_POS 2
#define  RECT_MENU_POS  1
```

这些标识符说明每个菜单模板中 Window 子菜单的位置。程序需要这些信息来通知客户窗口文档列表应出现在哪里。当然， MdiMenuInit 菜单没有 Window 子菜单，所以如前所述，文档列表应附加在第一个子菜单中(位置 0)。不过，实际上永远不会在此看到文档列表（在后面讨论此程序时，你可以发现这样做的原因）。

IDM_FIRSTCHILD 标识符不对应于菜单项，它与出现在 Window 子菜单上的文档列表中的第一个文档窗口相关联。这个识别符的值应当大于所有其它菜单 ID 的值。

18.3.2 程序初始化

在 MDIDEMO.C 中， WinMain 是从注册框架窗口和两个子窗口的窗口类开始的。窗口过程是 FrameWndProc 、 HelloWndProc 和 RectWndProc 。一般来说，这些窗口类应该与

不同的图标相关联。为了简单起见，我们将标准 IDI_APPLICATION 图标用于框架窗口和子窗口。

注意，我们已经定义了框架窗口类的 WNDCLASS 结构的 hbrBackground 域为 COLOR_APPWORKSPACE 系统颜色。由于框架窗口的客户区被客户窗口所覆盖并且客户窗口具有这种颜色，所以上面的定义不是完全必要的。但是，在最初显示框架窗口时，使用这种颜色似乎要好一些。

这三种窗口类中的 lpszMenuName 域都设置为 NULL。对 Hello 和 Rect 子窗口类来说，这是很自然的。对于框架窗口类，我在创建框架窗口时在 CreteWindow 函数中给出菜单句柄。

Hello 和 Rect 子窗口的窗口类将 WNDCLASS 结构中的 cbWndExtra 域置为非零值来为每个窗口分配额外空间，这个空间将用于存储指向一个内存块的指针（ HELLODATA 和 RECTDATA 结构的大小定义在 MDIDEMO.C 的开始处），这个内存块被用于存储每个文档窗口特有的信息。

下一步， WinMain 用 LoadMenu 装入三个菜单，并把它们的句柄保存到全局变量中。调用三次 GetSubMenu 函数可获得 Window 子菜单（文档列表将加在它上面）的句柄，同样也把它们保存到全局变量中。 LoadAccelerators 函数装入加速键表。

在 WinMain 中调用 CreatWindow 创建框架窗口。在 FrameWndProc 中处理 WM_CREATE 消息期间，框架窗口创建客户窗口。这项工作涉及到再一次调用函数 CreateWindow 。窗口类被设置为 MDICLIENT ，它是预先注册的 MDI 窗口类。CreateWindow 的最后一个参数必须设置为指向 CLIENTCREATESTRUCT 类型的结构的指针。此结构有两个域：

- hWindowMenu 是要加入文档列表的子菜单的句柄。在 MDIDEMO 中，它是 hMenuInitWindow ，是在 WinMain 期间获得的。后面将看到如何修改此菜单。
- idFirstChild 是与文档列表中的第一个文档窗口相关联的菜单标识符。它就是 IDM_FIRSTCHILD 。

再让我们回过头来看看 WinMain 。 MDIDEMO 显示新创建的框架窗口并进入消息循环。消息循环与正常的循环稍有不同：在调用 GetMessage 从消息队列中获得消息之后，MDI 程序把这条消息传送给 TranslateMDISysAccel(以及 TranslateAccelerator ，如果像 MDIDEMO 程序一样，程序本身也有菜单加速键的话)。

TranslateMDISysAccel 函数把可能对应于特定 MDI 加速键(例如 Ctrl-F6)的击键转换成 WM_SYSCOMMAND 消息。如果 TranslateMDISysAccel 或 TranslateAccelerator 都不返回 TRUE(表明某条消息已被这些函数之一转换)，就不能调用 TranslateMessage 和 DispatchMessage 。

注意传递到 TranslateMDISysAccel 和 TranslateAcclerator 的两个窗口句柄： hwndClient 和 hwndFrame 。 WinMain 函数通过用 GW_CHILD 参数调用 GetWindow 获得 hwndClient 窗口句柄。

18.4 创建子窗口

FrameWndProc 的大部分工作是用于处理通知菜单选择的 WM_COMMAND 消息。与平时一样，FrameWndProc 中的 wParam 参数包含着菜单 ID 号。

在 wParam 的值为 IDM_NEWHELLO 和 IDM_NEWRECT 的情况下，FrameWndProc 必须创建一个新的文档窗口。这涉及到初始化 MDICREATESTRUCT 结构中的域(大多数域对应于 CreateWindow 的参数)，并将消息 WM_MDICREATE 发送给客户窗口，消息的 lParam 参数设置为指向这个结构的指针。然后由客户窗口创建子窗口。FrameWndProc 可以直接调用 CreateMDIWindow 来创建子窗口。对于象 MDIDEMO 这样的单线程程序，你可以选择任何一种你喜欢的方式。

MDICREATESTRUCT 结构中的 szTitle 域一般是对应于文档的文件名。style 域设置为 WS_HSCROLL、WM_VSCROLL 或这两者的组合，以便在文档窗口中包括滚动条(不必调用 ShowScrollBar 来显示这些滚动条)。风格域也可以包括 WS_MINIMIZE 或 WS_MAXIMIZE，以便在最初时以最小化或最大化状态显示文档窗口。

MDICREATESTRUT 结构的 lParam 域为框架窗口和子窗口共享某些变量提供了一种方法。这个域可以设置为含有一个结构的内存块的内存句柄。在子文档窗口的 WM_CREATE 消息期间，lParm 是一个指向 CREATESTRUCT 结构的指针，这个结构的 lpCreateParams 域是一个指向用于创建窗口的 MDICREATESTRUCT 结构的指针。

客户窗口一旦接收到 WM_MDICREATE 消息就创建一个子文档窗口，并把窗口标题加到用于创建客户窗口的 MDICLIENTSTRUCT 结构中所指定的子菜单的底部。当 MDIDEMO 程序创建它的第一个文档窗口时，这个子菜单就是 MdiMenuInit 菜单中的 File 子菜单。后面将看到这个文档列表将如何移到 MdiMenuHello 和 MdiMenuRect 菜单的 Window 子菜单中。

菜单上可以列出 9 个文档，每个文档的前面是带有下划线的数字 1 至 9。如果创建的文档窗口多于 9 个，则这个清单后跟有“ More Windows ”菜单项。该项激活带有列表框的对话框，列表框列出了所有文档。这种文档列表的维护是 Windows 95 的 MDI 支持的最好的特性之一。

18.5 关于框架窗口的消息处理

在把注意力转移到子文档窗口之前，我们先继续讨论 FrameWndProc 的消息处理。

当从 File 菜单中选择 Close 时，MDIDEMO 关闭活动子窗口。它通过把 WM_MDIGETACTIVE 消息发送给客户窗口，而获得活动子窗口的句柄。如果子窗口以 WM_QUERYENDSESSION 消息来响应，那么 MDIDEMO 将 WM_MDIDESTROY 消息发送给客户窗口，从而关闭子窗口。

处理 File 菜单中的 Exit 选项只需要框架窗口过程给自己发送一条 WM_CLOSE 消息。

处理 Window 子菜单的 File 、 Arrange 、 Icons 选项是极容易的，只需把消息 WM_MDITILE 、 WM_MDICASCADE 和 WM_MDIICONARRANGE 发送给客户窗口。

处理 Close All 选项要稍微复杂一些。 FrameWndProc 调用 EnumChildWindows ，传送一个引用 CloseEnunProc 函数的指针。此函数把 WM_MDIRESTORE 消息发送给每个子窗口，紧跟着发出 WM_QUERYENDSESSION 和 WM_MDIDESTROY 。对图标平铺窗口来说并不就此结束，用 GW_OWNER 参数调用 GetWindow 时，返回的非 NULL 值可以表明这一点。

FrameWndProc 没有处理任何由 Color 菜单中对颜色的选择所导致的 WM_COMMAND 消息，这些消息应该由文档窗口负责处理。因此， FrameWndProc 把所有未经处理的 WM_COMMAND 消息发送到活动子窗口，以便子窗口可以处理那些与它们有关的消息。

框架窗口过程不予处理的所有消息都要送到 DefFrameProc ，它在框架窗口过程中取代了 DefWindowProc 。即使框架窗口过程捕获了 WM_MENUCHAR 、 WM_SETFOCUS 或 WM_SIZE 消息，这些消息也要被送到 DefFrameProc 中。

所有未经处理的 WM_COMMAND 消息也必须送给 DefFrameProc 。具体地说， FrameWndProc 并不处理任何 WM_COMMAND 消息，即使这些消息是用户在 Window 子菜单的文档列表中选择文档时产生的(这些选项的 wParam 值是以 IDM_FIRSTCHILD 开始的)。这些消息要传送到 DefFrameProc ，并在那里进行处理。

注意框架窗口并不需要维护它所创建的所有文档窗口的窗口句柄清单。如果需要这些句柄(如处理菜单上的 Closs All 选项时)，可以使用 EnumChildWindows 得到它们。

18.6 子文档窗口

现在看一下 HelloProc ，它是用于显示“ Hello, World!”的子文档窗口的窗口过程。

与用于多个窗口的窗口类一样，所有在窗口过程中定义的静态变量由基于该窗口类的所有窗口共享。

只有对每个唯一子窗口的数据才必须采用非静态变量的方法来存储。这样的技术要用到窗口属性。另一种方法（我使用的方法）是使用预留的内存空间；可以在注册窗口类时将 WNDCLASSEX 结构的 cbWndExtra 域置为非零值以便预留这部分内存空间。

MDIDEMO 程序使用这个内存空间来保存一个指针，这个指针指向一块与 HELLODATA 结构大小相同的内存块。在处理 WM_CREATE 消息时， HelloWndProc 分配这块内存，初始化它的两个域(它们用于指定当前选中的菜单项和文本颜色)，并用 SetWindowLong 将内存指针保存到预留的空间中。

当处理改变文本颜色的WM_COMMAND消息(回忆一下，这些消息来自框架窗口过程)时， HelloWndProc 用 GetWindowLong 获得包含 HELLODATA 结构的内存块的指针。利用这个结构， HelloWndProc 清除原来对菜单项的选择，设置所选菜单项为选中状态，并保存新的颜色。

当窗口变成活动或不活动的时候，文档窗口过程都会收到 WM_MDIACTIVATE 消息(lParam 的值是否是这个窗口的句柄表明了该窗口是活动的还是不活动的)。你也许还能记

起 MDIDEMO 程序中有三个不同的菜单：当无文档时为 MdiMenuInit；当 Hello 文档窗口是活动窗口时为 MdiMenuHello；当 Rect 文档窗口为活动窗口时为 MdiMenuRect。

WM_MDIACTIVATE 消息为文档窗口提供了一个修改菜单的机会。如果 lParam 中含有本窗口的句柄（意味着本窗口将变成活动的），那么 HelloWndProc 就将菜单改为 MdiMenuHello。如果 lParam 中包含另一个窗口的句柄，那么 HelloWndProc 将菜单改为 MdiMenuInit。

HellWndProc 通过把 WM_MDISETMENU 消息发送给客户窗口来修改菜单，客户窗口通过从当前菜单上删除文档列表并把它添加到一个新的菜单上来处理这条消息。这就是文档列表从 MdiMenuInit 菜单(它在创建第一个文档时有效)传送到 MdiMenuHello 菜单的方法。在 MDI 应用程序中不要用 SetMenu 函数改变菜单。

另一项工作涉及到 Color 子菜单上的选中标志。像这样的程序选项对每个文档来说都是不同的，例如，可以在一个窗口中设置黑色文本，在另一个窗口中设置红色文本。菜单选中标志应能反映出活动窗口中选择的选项。由于这种原因，HellWndProc 在窗口变成非活动窗口时清除选中菜单项的选中标志，而当窗口变成活动窗口时设置适当菜单项的选中标志。

WM_MDIACTIVATE 的 wParam 和 lParam 值分别是失活和被激活窗口的句柄。当窗口被创建时，窗口过程得到的第一条 WM_MDIACTIVATE 消息的 lParam 参数被设置为当前窗口的句柄。而当窗口被消除时，窗口过程得到的此消息的 lParam 参数被设置为另一个值。当用户从一个文档切换到另一个文档时，前一个文档窗口收到一条 WM_MDIACTIVATE 消息，其 wParam 参数为它自己的句柄(此时，窗口过程将菜单置为 MdiMenuInit)；后一个文档窗口收到一条 WM_MDIACTIVATE 消息，其 lParam 参数是它自己的句柄(此时，窗口过程将菜单设置为 MdiMenuHello 或 MdiMenuRect 中适当的那个)。如果关闭所有的窗口，那么菜单将保持为 MdiMenuInit。

当用户从菜单中选择 Close 或 Close All 时，FrameWndPrce 给子窗口发送一条 WM_QUERYENDSESSION 消息。HelloWndProc 将显示一个消息框并询问用户是否要关闭窗口，以此来处理 WM_QUERYENDSESSION 和 WM_CLOSE 消息(在真实的应用程序中，消息框会询问是否需要保存文件)。如果用户表示不能关闭窗口，那么窗口过程返回 0。

在 WM_DESTROY 消息期间，HelloWndProc 释放在 WM_CREATE 消息期间分配的内存块。

所有未经处理的消息必须传送到用于缺省处理的 DefMDIChildProc(不是 DefWindowProc)。不论子窗口过程是否使用了这些消息，有几个消息必须被传送给 DefMDIChildProc。这些消息是：WM_CHELDACTIVATE、WM_GETMINMAXINFO、WM_MENUCHAR、WM_MOVE、WM_SETFOCUS、WM_SIZE 和 WM_SYSCOMMAND。

RectWndProc 与 HelloWndProc 非常相似，但是它比 HelloWndProc 还要简单一些（不含菜单选项并且无需用户确认是否关闭窗口），所以这里不对它进行讨论了。但应该注意到，在处理 WM_SIZE 之后 RectWndProc 使用了 break 语句，所以 WM_SIZE 消息被传给了 DefMDIChildProc。

18.7 结束处理

在 WinMain 中，MDIDEMO 使用 LoadMenu 装入资源描述文件中定义的三个菜单。一般说来，当菜单所在的窗口被清除时，Windows 也要清除与之关联的菜单。对于 MdiMenuInit 菜单，这就够了。而那些没有联系到窗口的菜单（如 MDIDEMO 中的 Hello 和 Rect 菜单）将继续占用宝贵的内存空间，甚至在程序结束之后仍然如此。由于这个原因，MDIDEMO 在 WinMain 的末尾调用了两次 DestroyMenu 来清除 Hello 和 Rect 菜单。

18.8 窗口过程的强大功能

Windows 95 对多文档界面的支持大多被封装在 MDICLIENT 窗口类中。这已充分地说明了 Windows 面向对象的体系结构的强大功能。客户窗口过程作为框架窗口和各种文档窗口之间的中间层而发挥作用。

现在，让我们来看看 Windows 的另一强大功能特性——动态链接库（DLL）。

第 19 章　动态链接库

动态链接库（也称为 DLL 、动态库、“ dynalink ”库或库模块）是 Windows 最重要的组成要素之一。大多数与 Windows 相关的磁盘文件要么是程序模块，要么是动态链接库模块。迄今为止，我们都是在编写 Windows 应用程序；现在是尝试编写动态链接库的时候了。许多你已经学会的编写应用程序的规则同样适用于编写这些库模块，但也有一些重要的不同。

19.1　库的基本知识

正如前面所看到的， Windows 应用程序是一个可执行文件，它通常创建一个或几个窗口，并使用消息循环接收用户输入。通常，动态链接库并不能直接执行，也不接收消息。它们是一些独立的文件，其中包含能被程序或其它 DLL 调用来完成某项工作的函数。只有在其它模块调用动态链接库中的函数时，它才发挥作用。

所谓“动态链接”，是指 Windows 把一个模块中的函数调用链接到库模块中的实际函数上的过程。在程序开发中，你将各种目标模块(.OBJ)、运行库(.LIB)文件，以及经常有的一个编译好的资源(RES)文件链接在一起，以便创建 Windows 的.EXE 文件，这时的链接是“静态链接”。动态链接与此不同，它发生在运行时刻。

KERNEL32.DLL 、 USER32.DLL 和 GDI32.DLL 、各种驱动程序文件如 KEYBOARD.DRV 、 SYSTEM.DRV 、 MOUSE.DRV 和视频及打印机驱动程序都是动态链接库。这些库都能被所有 Windows 应用程序使用。

有些动态链接库（如字体文件等）被称为是“纯资源”。它们只包含数据（通常是资源的形式）而不包含代码。由此可见，动态链接库的目的之一就是提供能被许多不同的应用程序所使用的函数和资源。在常规的操作系统中，只有操作系统本身才包含其它应用程序能够调用来完成某一工作的例程。在 Windows 中，一个模块调用另一个模块函数的过程被推广了。结果使得编写一个动态链接库，也就是在扩充 Windows 。当然，也可认为动态链接库（包括构成 Windows 的那些库例程）是对用户程序的扩充。

尽管一个动态链接库模块可能有其它扩展名（如.EXE 或.FON ），但 Microsoft Windows95 中的标准扩展名是.DLL 。只有带.DLL 扩展名的动态链接库能被 Widows 自动加载。如果文件有其它扩展名，则程序必须显式地使用 LoadLibrary 函数加载该模块。

你通常会发现，动态链接库在大型应用中最有意义。例如，假设要为 Windows 编写一个由几个不同的程序组成的大型财务软件包，就会发现这些应用程序会使用许多共同的例程。可以把这些公共例程放入一个常规的目标库（带.LIB 扩展名）中，并在使用 LINK

静态链接时把它们加入各程序模块中。但这种方法是很浪费的，因为软件包中的每个程序都包含与公共例程相同的代码。而且，如果修改了库中的某个例程，就要重新链接使用此例程的所有程序。然而，如果把这些公共例程放到称为 ACCOUNT.DLL 的动态链接库中，就可解决这两个问题。只有库模块才包含所有程序都要用到的例程（这样能为存储文件节省磁盘空间，并且在同时运行多个应用程序时节省内存），并且，可以修改库模块而不用重新链接各个程序。

动态链接实际上是一种可以独立自存的产品。例如，假设你编写了一系列三维绘图例程，并把它们放入动态链接库 GDI3.DLL 中。如果其它软件开发者对此库很感兴趣，你就可以授权他们将其加入他们的各种图形程序中。使用多个这样的图形程序的用户只需要一个 GDI3.DLL 文件。

19.1.1 库：一词多义

关于动态链接库所存在的模糊认识，部分原因是由于“库”这个词被用在几种不同的上下文中。除了动态链接库之外，我们也用它来称呼“目标库(object libraries)”或“输入库（ import libraries ）”。

目标库是带.LIB 扩展名的文件。在用链接程序进行静态链接时，它的代码就会加到程序的.EXE 文件中。例如，在 Microsoft Visual C++中，同程序链接的常规 C 运行目标库被称为 LIBC.LIB 。

输入库是目标库文件的一种特殊形式。像目标库一样，输入库有.LIB 扩展名，并且被链接器用来确定（ resolve ）源程序代码中的函数调用。但输入库不含代码，而是为链接程序提供信息，以便在.EXE 文件中建立动态链接时要用到的重定位表。包含在 Microsoft 编译器中的 KERNEL32.LIB 、 USER32.LIB 和 GDI32.LIB 文件是 Windows 函数的输入库。如果一个程序调用 Rectangle ， GDI32.LIB 将告诉 LINK ，该函数在 GDI32.DLL 动态链接库中。该信息被记录在.EXE 文件中，使得程序运行时， Windows 能够和 GDI32.DLL 动态链接库进行动态链接。

目标库和输入库只用在程序开发期间，而动态链接库在运行期间使用。当一个使用动态链接库的程序运行时，该动态链接库必须在磁盘上。当 Windows 95 需要运行一个使用了动态链接库的程序而需要装载该库时，库文件必须存储在含有该.EXE 程序的目录下，或者是在当前目录下，或 Windows 系统目录下，或 Windows 目录下，或者是在通过 MS-DOS 环境中的 PATH 可以访问到的目录下（这些目录将被顺序搜索到）。

19.2 一个简单的 DLL

和平常一样，我们将从简单的入手。图 19-1 给出了只包含一个函数，被称为 EDRLIB 的动态链接库的源代码。这个文件名中的“ EDR ”表示“简单绘图例程（ easy drawing routine ）”，你可以很容易地在其中增加其它的函数，以简化你的应用中的绘图函数。在 EDRLIB 中的那个唯一的函数被称为 EdrCenterText ，它在一个矩形中间部分画出一个以

NULL 结尾的文本串，非常像 DrawText，只是没有那么多参数。

EDRLIB.MAK

```
#----------------------
# EDRLIB.MAK make file
#----------------------

edrlib.dll : edrlib.obj
     $(LINKER) $(DLLFLAGS) -OUT:edrlib.dll edrlib.obj $(GUILIBS)

edrlib.obj : edrlib.c edrlib.h
     $(CC) $(CFLAGS) edrlib.c
```

EDRLIB.H

```
/*----------------------
   EDRLIB.H header file
  ----------------------*/

#define EXPORT extern "C" __declspec (dllexport)

EXPORT BOOL CALLBACK EdrCenterText (HDC, PRECT, PSTR) ;
```

EDRLIB.C

```
/*------------------------------------------------
   EDRLIB.C -- Easy Drawing Routine Library module
               (c) Charles Petzold, 1996
  ------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "edrlib.h"

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     return TRUE ;
     }

EXPORT BOOL CALLBACK EdrCenterText (HDC hdc, PRECT prc, PSTR pString)
     {
     int  iLength ;
     SIZE size ;

     iLength = strlen (pString) ;

     GetTextExtentPoint32 (hdc, pString, iLength, &size) ;

     return TextOut (hdc, (prc->right - prc->left - size.cx) / 2,
                          (prc->bottom - prc->top - size.cy) / 2,
                     pString, iLength) ;
     }
```

图 19-1 EDRLIB库

你可以看到，和我们以前对应用程序所使用的 make 文件相比，这里的 make 文件有两个不同之处。首先，在链接命令行上，我用环境变量 DLLFLAGS 代替了 GUIFLAGS 。这个变量包含在第 1 章中介绍的 MSC.BAT 文件中，它只是包含了链接命令行参数-DLL 。这个参数告诉链接器结果文件将是一个动态链接库。其次，你还将注意到 make 文件创建的是一个名为 EDRLIB.DLL 的文件，而不是一个.EXE 文件。

EDRLIB.H 文件首先定义了一个名为 EXPORT 的宏：

```
#define EXPORT extern "C"  __declspec (dllexport)
```

在动态链接库中定义函数时使用 EXPORT 关键字，将告诉链接器该函数可以被其它程序使用。 EdrCenterText 函数定义在头文件中并使用了这个 EXPORT 定义。函数还用了 CALLBACK 进行定义，这一点与窗口过程一样。

EDRLIB.C 看起来和普通的 Windows C 文件也稍有不同，它包含一个名为 DllMain 而不是 WinMain 的函数。这个函数被用于进行初始化和解初始化(de-initialization)，我将在本章后面进行讨论。针对我们的目的，目前我们所需要做的是返回 TRUE 。最后， EDRLIB.C 包含 EdrCenterText 函数。

在命令行上运行

```
NMAKE EDRLIB.MAK
```

将创建动态链接库 EDRLIB.DLL 。你还将发现另外两个新文件。 EDRLIB.LIB 是一个我们马上将用到的输入库。 EDRLIB.EXP 是一个输出库，它是链接过程的一个副产物。你可以删掉它。

EDRLIB.DLL 可以工作吗？让我们试一试。图 19-2 中的 EDRTEST 使用 EDRLIB.DLL 在其客户区内居中显示一个文本串。

EDRTEST.MAK

```
#-----------------------
# EDRTEST.MAK make file
#-----------------------

edrtest.exe : edrtest.obj edrlib.lib
     $(LINKER) $(GUIFLAGS) -OUT:edrtest.exe edrtest.obj edrlib.lib $(GUILIBS)

edrtest.obj : edrtest.c edrlib.h
     $(CC) $(CFLAGS) edrtest.c
```

EDRTEST.C

```
/*---------------------------------------------------
   EDRTEST.C -- Program using EDRLIB dynamic link library
                (c) Charles Petzold, 1996
  ---------------------------------------------------*/

#include <windows.h>
#include "edrlib.h"
```

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "StrProg" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

     switch (iMsg)
          {
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               GetClientRect (hwnd, &rect) ;
```

```
            EdrCenterText (hdc, &rect,
                           "This string was displayed by a DLL") ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

图 19-2 EDRTEST程序

这看起来像一个普通的 Windows 程序，而它的确也是。然而，注意 EDRTEST.C 包含了 EDRLIB.H 来定义 WM_PAINT 消息处理中调用的 EdrCenterText 函数。还应该注意 EDRTEST.MAK 在 link 语句中包含了 EDRLIB.LIB 文件。输入库为 LINK 提供了建立 EDRTEST.EXE 文件时引用 EDRLIB.DLL 文件所需要的信息。

认识到EdrCenterText代码没有被包含在EDRTEST.EXE文件中是十分重要的。事实上，EDRTEST.EXE 中只有对 EDRLIB.DLL 文件和 EdrCenterText 的一个引用。 EDRTEST.EXE 需要 EDRLIB.DLL 才能运行。

在 EDRTEST.C 源代码文件中包含 EDRLIB.H 相当于包含 WINDOWS.H 。在 link 命令行中包含 EDRLIB.LIB 相当于包含所有 Windows 输入库（诸如 USER32.LIB ），这些输入库在 make 文件所引用的 GUILIBS 环境变量中列出。当你的程序运行时，它采用和链接 USER32.DLL 一样的方式链接 EDLIB.DLL 。恭喜！你已经创建了 Windows 95 的一种扩展！

在我们继续下去之前，还有几句关于动态链接库这一主题的话。虽然我刚才将一个 DLL 视为 Windows 95 的一种扩展，但是它也是对你的应用程序的一种扩展。 DLL 所做的任何事情都是代表相应的应用程序做的。例如，它分配的所有内存都为应用程序所拥有，它创建的所有窗口都为应用程序所拥有，它打开的所有文件都为应用程序所拥有。多个应用程序可以同时使用相同的 DLL ，但是在 Windows 95 下，这些应用程序避免了相互干扰。然而，如果你编写的 DLL 中包含可以从一个程序的多个线程中调用的函数，那么你应该使用 CFLAGSMT 环境变量而不是 CFLAGS 来进行编译。

由于代码是不可写的，所以不同的进程能够共享一个动态链接库中的同样代码。但是，由一个 DLL 维护的数据对不同进程是不同的。每个进程有它自己的地址空间用于 DLL 使用的数据。在进程之间共享内存要求额外的一些工作（我们将会看到）。

19.3 DLL 中的共享内存

Windows 95 将同时使用同一动态链接库的应用隔离开来，这很好。然而，有时这并不是所希望的情况。你可能想编写一个 DLL ，其中包含一些可以在多个应用，或者同一应用的多个实例中共享的内存。这涉及到共享内存（实际上是一个内存映象文件）的使用，关

于共享内存，我曾经在第 13 章中提到过并保证将在后面进行演示。

让我们通过一个叫做 STRPROG（“串程序”）的程序和一个叫做 STRLIB（“串库”）的动态链接库来看一看这是如何工作的。 STRLIB 有三个由 STRPROG 调用的输出函数。仅仅为了增加趣味， STRLIB 中的一个函数使用了 STRPROG 中定义的一个回调函数。

STRLIB 是一个存储和排序多达 256 个字符串的动态链接库模块。这些串由大写字母组成，并存储在 STRLIB 自己的数据段中。 STRPROG 能分别使用 STRLIB 中的三个函数来添加或删除串，以及获取 STRLIB 中的所有当前串。该程序有两个菜单项（ Enter 和 Delete ）， 它们可以激活对话框来增加或删除串。 STRPROG 在 STRPROG 的客户区中显示 STRLIB 的数据段中保存的所有当前串。

STRLIB 中定义的以下函数在 STRLIB 的共享内存中增加一个串:

```
EXPORT BOOL CALLBACK AddString (PSTR pStriingIn)
```

参数 pStringIn 是指向该串的指针，该串在 AddString 函数中被转换成大写字母。如果在 STRLIB 数据段中已存在等价串，则这个函数增加该串的一份拷贝。成功的 AddString 返回 TRUE(非 0)否则返回 FALSE 。如果串的长度为 0 ，或者不能分配用以保存串的内存，或者已经保存了 256 个串，都可能导致返回值为 FALSE 。

STRLIB 函数可以从 STRLIB 的共享内存中删除一个串:

```
EXPORT BOOL CALLBACK DeleteString (PSTR pStringIn)
```

同样，参数 pStringIn 是指向该串的指针。如果存在多个匹配的串， 则只删除第一个匹配串。成功时 DeleteString 返回 TRUE(非零)，否则返回 FALSE 。如果串的长度为零或者找不到匹配的串，则返回 FALSE 。

这个 STRLIB 函数使用调用它的程序中的一个回调函数来枚举 STRLIB 的共享内存中的当前串:

```
EXPORT int CALLBACK GetStrings (PSTRCB pfnGetStrCallBack, PVOID pParam )
```

该回调函数必须如下定义:

```
EXPORT BOOL CALLBACK GetStrCallBack (PSTR pString ,  PVOID pParam)
```

GetStrings 的 pfnGetStrCallBack 参数指向回调函数。 GetStrings 为每个串调用一次 GetStrCallBack 直到回调函数返回 FALSE （ 0 ）。 GetStrings 返回传递给回调函数的串的数目。 pParam 参数是指向程序员自定义数据的一个指针。

19. 3. 1　STRLIB 库

图 19-3 给出了创建 STRLIB.DLL 动态链接库模块所必要的三个文件。

STRLIB.MAK

```
#----------------------
# STRLIB.MAK make file
#----------------------
```

```
strlib.dll : strlib.obj
     $(LINKER) $(DLLFLAGS) -SECTION:shared,rws -OUT:strlib.dll \
               strlib.obj $(GUILIBS)

strlib.obj : strlib.c strlib.h
     $(CC) $(CFLAGS) strlib.c
```

STRLIB.H

```
/*----------------------
   STRLIB.H header file
  ----------------------*/

typedef BOOL (CALLBACK *PSTRCB) (PSTR, PVOID) ;
#define MAX_STRINGS 256

#define EXPORT  extern "C" __declspec (dllexport)

EXPORT BOOL CALLBACK AddString    (PSTR) ;
EXPORT BOOL CALLBACK DeleteString (PSTR) ;
EXPORT int  CALLBACK GetStrings   (PSTRCB, PVOID) ;
```

STRLIB.C

```
/*------------------------------------------------
   STRLIB.C -- Library module for STRPROG program
               (c) Charles Petzold, 1996
  ------------------------------------------------*/

#include <windows.h>
#include "strlib.h"

#pragma data_seg ("shared")

PSTR pszStrings[MAX_STRINGS] = { NULL } ;
int  iTotal = 0 ;

#pragma data_seg ()

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     int i ;

     switch (fdwReason)
          {
          // Nothing to do when process (or thread) begins

          case DLL_PROCESS_ATTACH :
          case DLL_THREAD_ATTACH :
          case DLL_THREAD_DETACH :
               break ;

          // When process terminates, free any remaining blocks

          case DLL_PROCESS_DETACH :
```

```
            for (i = 0 ; i < iTotal ; i++)
                UnmapViewOfFile (pszStrings[i]) ;
            break ;
        }

    return TRUE ;
    }

EXPORT BOOL CALLBACK AddString (PSTR pStringIn)
    {
    HANDLE hString ;
    PSTR   pString ;
    int    i, iLength, iCompare ;

    if (iTotal == MAX_STRINGS - 1)
        return FALSE ;

    iLength = strlen (pStringIn) ;
    if (iLength == 0)
        return FALSE ;

    hString = CreateFileMapping ((HANDLE) -1, NULL, PAGE_READWRITE,
                                 0, 1 + iLength, NULL) ;
    if (hString == NULL)
        return FALSE ;

    pString = (PSTR) MapViewOfFile (hString, FILE_MAP_WRITE, 0, 0, 0) ;
    strcpy (pString, pStringIn) ;
    AnsiUpper (pString) ;

    for (i = iTotal ; i > 0 ; i--)
        {
        iCompare = strcmpi (pStringIn, pszStrings[i - 1]) ;

        if (iCompare >= 0)
            break ;

        pszStrings[i] = pszStrings[i - 1] ;
        }

    pszStrings[i] = pString ;

    iTotal++ ;
    return TRUE ;
    }

EXPORT BOOL CALLBACK DeleteString (PSTR pStringIn)
    {
    int i, j, iCompare ;

    if (0 == strlen (pStringIn))
        return FALSE ;

    for (i = 0 ; i < iTotal ; i++)
        {
```

```
          iCompare = lstrcmpi (pszStrings[i], pStringIn) ;

          if (iCompare == 0)
               break ;
          }

     // If given string not in list, return without taking action

     if (i == iTotal)
          return FALSE ;

     // Else free memory occupied by the string and adjust list downward

     UnmapViewOfFile (pszStrings[i]) ;

     for (j = i ; j < iTotal ; j++)
          pszStrings[j] = pszStrings[j + 1] ;

     pszStrings[iTotal--] = NULL ;     // Destroy unused pointer
     return TRUE ;
     }

EXPORT int CALLBACK GetStrings (PSTRCB pfnGetStrCallBack, PVOID pParam)
     {
     BOOL bReturn ;
     int  i ;

     for (i = 0 ; i < iTotal ; i++)
          {
          bReturn = pfnGetStrCallBack (pszStrings[i], pParam) ;

          if (bReturn == FALSE)
               return i + 1 ;
          }
     return iTotal ;
     }
```

图 19-3 STRLIB库

19.3.2 库入口/出口点

你可以在 STRLIB.C 中看到，现在我们利用了一下 DllMain 。当库首次启动和结束时，调用这个函数。虽然我也在 EDRLIB.C 中包含了一个 DllMain 函数，但它其实是不必要的；缺省地，链接器会包含一个具有同样功能的函数。

DllMain 的第一个参数是库的实例句柄。如果你的库用到需要实例句柄（诸如 DialogBox ）的资源，那么你应该将 hInstance 保存为一个全局变量。 DllMain 的最后一个参数由系统保留。

fdwReason 参数可以是四个值之一，说明为什么 Windows 95 要调用 DllMain 函数。在下面的讨论中，请记住一个程序可以被多次装入，并在 Windows 下一起运行。每当一个程序被装入时，它都被认为是一个独立的进程。

fdwReason 的一个值 DLL_PROCESS_ATTACH 表示动态链接库被映射到一个进程的地址空间。链接库可以根据这个线索进行初始化，为以后来自该进程的请求提供服务。例如，这类初始化可能包括内存分配。在一个进程的生命周期内，只有一次对 DllMain 的调用以 DLL_PROCESS_ATTACH 为参数。使用同一动态链接库的其它任何进程都将导致另一个使用 DLL_PROCESS_ATTACH 参数的 DllMain 调用，但这是对新进程的调用。

如果初始化成功， DllMain 应该返回一个非零值。返回零将导致 Windows 不运行该程序。

当 fdwReason 的值为 DLL_PROCESS_DETACH 时，意味着进程不再需要动态链接库了，从而提供给库自己清除自己的机会。在 Windows 95 下，这种处理并不是严格必须的，但这是一种好的编程习惯。

类似地，当以 DLL_THREAD_ATTACH 为 fdwReason 参数调用 DllMain 时，意味着某个进程创建了一个新的线程。当线程中止时， Windows 以 DLL_THREAD_DETACH 为 dwReason 参数调用 DllMain 。请注意，如果动态链接库是在线程被创建之后和一个进程链接的，那么可能会得到一个没有相应的先行 DLL_THREAD_ATTACH 调用的 DLL_THREAD_DETACH 调用。

当使用一个 DLL_THREAD_DETACH 参数调用 DllMain 时，线程仍然存在。动态链接库甚至可以在这个过程期间发送线程消息。但是它不应该使用 PostMessage ，因为线程可能在此消息被检索到之前已经退出了。

除了 DllMain 函数之外， STRLIB 只包含了三个输出给其它程序使用的函数。这三个函数被定义为 EXPORT ，导致 LINK 将它们列在 STRLIB.LIB 输入库中。

19.3.3 STRPROG 程序

STRPROG 程序如图 19-4 所示，其内容是相当直观的。两个菜单选项（ Enter 和 Delete ）激活一个对话框，让你输入一个串，然后 STRPROG 调用 AddString 或者 DeleteString 。当程序需要更新它的客户区时，调用 GetStrings 并使用函数 GetStrCallBack 来列出所枚举的串。

STRPROG.MAK

```
#-----------------------
# STRPROG.MAK make file
#-----------------------

strprog.exe : strprog.obj strprog.res strlib.lib
     $(LINKER) $(GUIFLAGS) -OUT:strprog.exe strprog.obj strprog.res \
               strlib.lib $(GUILIBS)

strprog.obj : strprog.c strprog.h strlib.h
     $(CC) $(CFLAGS) strprog.c

strprog.res : strprog.rc strprog.h
     $(RC) $(RCVARS) strprog.rc
```

STRPROG.C

```
/*-------------------------------------------------------
   STRPROG.C -- Program using STRLIB dynamic link library
                (c) Charles Petzold, 1996
  -------------------------------------------------------*/

#include <windows.h>
#include <string.h>
#include "strprog.h"
#include "strlib.h"

#define MAXLEN 32
#define WM_DATACHANGE WM_USER

typedef struct
     {
     HDC  hdc ;
     int  xText ;
     int  yText ;
     int  xStart ;
     int  yStart ;
     int  xIncr ;
     int  yIncr ;
     int  xMax ;
     int  yMax ;
     }
     CBPARAM ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[] = "StrProg" ;
char szString[MAXLEN] ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;
```

```
     hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

BOOL CALLBACK DlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     switch (iMsg)
          {
          case WM_INITDIALOG :
               SendDlgItemMessage (hDlg, IDD_STRING, EM_LIMITTEXT,
                                   MAXLEN - 1, 0) ;
               return TRUE ;

          case WM_COMMAND :
               switch (wParam)
                    {
                    case IDOK :
                         GetDlgItemText (hDlg, IDD_STRING, szString, MAXLEN) ;
                         EndDialog (hDlg, TRUE) ;
                         return TRUE ;

                    case IDCANCEL :
                         EndDialog (hDlg, FALSE) ;
                         return TRUE ;
                    }
          }
     return FALSE ;
     }

BOOL CALLBACK EnumCallBack (HWND hwnd, LPARAM lParam)
     {
     char szClassName[16] ;

     GetClassName (hwnd, szClassName, sizeof (szClassName)) ;

     if (0 == strcmp (szClassName, szAppName))
          SendMessage (hwnd, WM_DATACHANGE, 0, 0) ;

     return TRUE ;
     }
```

```
BOOL CALLBACK GetStrCallBack (PSTR pString, CBPARAM *pcbp)
     {
     TextOut (pcbp->hdc, pcbp->xText, pcbp->yText,
              pString, strlen (pString)) ;

     if ((pcbp->yText += pcbp->yIncr) > pcbp->yMax)
          {
          pcbp->yText = pcbp->yStart ;
          if ((pcbp->xText += pcbp->xIncr) > pcbp->xMax)
               return FALSE ;
          }
     return TRUE ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE  hInst ;
     static int        cxChar, cyChar, cxClient, cyClient ;
     CBPARAM           cbparam ;
     HDC               hdc ;
     PAINTSTRUCT       ps ;
     TEXTMETRIC        tm ;

     switch (iMsg)
          {
          case WM_CREATE :
               hInst = ((LPCREATESTRUCT) lParam)->hInstance ;
               hdc   = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = (int) tm.tmAveCharWidth ;
               cyChar = (int) (tm.tmHeight + tm.tmExternalLeading) ;
               ReleaseDC (hwnd, hdc) ;

               return 0 ;

          case WM_COMMAND :
               switch (wParam)
                    {
                    case IDM_ENTER :
                         if (DialogBox (hInst, "EnterDlg", hwnd, &DlgProc))
                              {
                              if (AddString (szString))
                                   EnumWindows (&EnumCallBack, 0) ;
                              else
                                   MessageBeep (0) ;
                              }
                         break ;

                    case IDM_DELETE :
                         if (DialogBox (hInst, "DeleteDlg", hwnd, &DlgProc))
                              {
                              if (DeleteString (szString))
                                   EnumWindows (&EnumCallBack, 0) ;
                              else
                                   MessageBeep (0) ;
```

```
                    }
                break ;
            }
            return 0 ;

        case WM_SIZE :
            cxClient = (int) LOWORD (lParam) ;
            cyClient = (int) HIWORD (lParam) ;
            return 0 ;

        case WM_DATACHANGE :
            InvalidateRect (hwnd, NULL, TRUE) ;
            return 0 ;

        case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps) ;

            cbparam.hdc    = hdc ;
            cbparam.xText  = cbparam.xStart = cxChar ;
            cbparam.yText  = cbparam.yStart = cyChar ;
            cbparam.xIncr  = cxChar * MAXLEN ;
            cbparam.yIncr  = cyChar ;
            cbparam.xMax   = cbparam.xIncr * (1 + cxClient / cbparam.xIncr) ;
            cbparam.yMax   = cyChar * (cyClient / cyChar - 1) ;

            GetStrings ((PSTRCB) GetStrCallBack, (PVOID) &cbparam) ;

            EndPaint (hwnd, &ps) ;
            return 0 ;

        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

STRPROG.RC

```
/*-----------------------------
   STRPROG.RC resource script
  -----------------------------*/

#include <windows.h>
#include "strprog.h"

StrProg MENU
     {
     MENUITEM  "&Enter!",  IDM_ENTER
     MENUITEM  "&Delete!", IDM_DELETE
     }

EnterDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
     {
```

```
     LTEXT          "&Enter:",  0,            4,  8,  24,  8
     EDITTEXT                   IDD_STRING,  32,  6, 154, 12
     DEFPUSHBUTTON  "Ok",       IDOK,        44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,   114, 24,  32, 14
     }

DeleteDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
     {
     LTEXT          "&Delete:", 0,            4,  8,  28,  8
     EDITTEXT                   IDD_STRING,  36,  6, 150, 12
     DEFPUSHBUTTON  "Ok",       IDOK,        44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,   114, 24,  32, 14
     }
```

STRPROG.H

```
/*-----------------------
   STRPROG.H header file
  -----------------------*/

#define IDM_ENTER      1
#define IDM_DELETE     2
#define IDD_STRING  0x10
```

图 19-4 STRPORG程序

STRPROG.C 包含 STRPROG.H 头文件，其中只是定义了在 STRPROG.RC 资源描述文件中使用的常数。它还包含了 STRLIB.H 头文件，其中定义了 STRPROG 将使用的 STRLIB 中的三个函数。

19.3.4 运行 STRPROG

建立了 STRLIB. DLL 和 STRPROG. EXE 之后，就可以运行 STRPROG 了。在运行 STRPROG 之前，要确认 STRLIB.DLL 在当前目录或 Windows 可以访问到的目录（前面所讨论的）中。当你运行 STRPROG 时 Windows 必须能加载 STRLIB. DLL 。如果 Windows 找不到 STRLIB.DLL，它将显示一个消息框来报告此问题。

当你执行 STRPROG.EXE 时，Windows 完成对外部库模块中函数的定位（fix-ups），这些函数中的大多数位于普通的 Windows 动态链接库中。但是 Windows 也会发现程序调用了三个来自 STRLIB 的函数，因此 Windows 将 STRLIB.DLL 装入内存中并调用 STRLIB 的初始化例程。在 STRPROG 内对这三个函数的调用被动态地链接到 STRLIB 中的函数。然后，用户就可以使用 STRPROG 向(从)STRLIB 的内部表中增加(删除)串了。

从 STRPROG 中调用 STRLIB 中的 AddString 、 DeleteString 和 GetStrings 函数的效率是很高的，并不比调用任何别的外部模块增加什么开销。实际上，STRPROG 与 STRLIB 之间的链接效率，就如同 STRLIB 中的这三个函数本身就处于 STRPROG 中一样。你或许会说：“那又怎样呢？我何苦要创建动态链接库呢？难道我不能将这三个例程包含到 STRPROG.EXE 中吗？”

你当然能。从某种意义上讲，STRLIB 只是 STRPROG 的扩充。然而，用户可能有兴趣看一看执行 STRPROG 的第二个实例时会是什么情况。STRLIB 将字符串和它们的指针存储在共享内存中，允许 STRPROG 的所有实例共享这些数据。让我们来看一看这是如何做到的。

19.3.5　在 STRPROG 实例之间共享数据

Windows 在一个 Win32 进程的地址空间周围筑了一道墙。通常，一个进程的地址空间中的数据是私有的，对别的进程是不可见的。但是运行 STRPROG 的多个实例表明了 STRLIB 在程序的所有实例之间共享数据是毫无问题的。当你在一个 STRPROG 窗口中增加或者删除一个串时，这种改变将立即反映在其它的窗口中。

STRLIB 共享两种类型的数据：串和串指针。作为例子，这两种类型各使用一种不同的共享方式。我们已经在第 13 章中遇到过其中一种。STRLIB 在内存映象文件中存储每个串，从而使这些串可为所有进程所见。

对于串指针，STRLIB 将它们保存在作共享用的一个特殊内存段中：

```
#pragma data_seg ("shared")
PSTR pszString[MAX_STRING] = {NULL};
int iTotal = 0;
#pragma data_seg ( )
```

第一个#pragma 语句创建数据段，这里命名为 shared 。你可以将这段命名为任何一个你喜欢的名字，但是链接器只认识前 8 个字符。在#pragma 之后的所有初始化了的变量都进入 shared 段中。第二个#pragma 语句标志段的结束。对变量进行专门的初始化是很重要的；否则编译器将把它们放在普通的未初始化的段中而不是放在 shared 中。

链接器必须知道有一个“ shared ”段。在链接器的命令行，指定-SECTION 参数，如下所示：

```
-SECTION:shared,rws
```

“ rws ”字母指示段具有读、写和共享的属性。

现在 STRPROG 的所有实例看到的将是串和指针的一个实例。在 STRPROG 中的 EnumCallBack 函数负责通知所有 STRPROG 的实例：STRLIB 数据段中的内容发生了改变。EnuWindows 导致 Windows 使用所有父窗口句柄调用 EnumCallBack 。EnumCallBack 接着检查每个窗口的类名是否等于“ StrProg ”；如果等于，则发送一个自定义的 WM_DATACHANGE 消息给该窗口。你可以想象 STRLIB 的一种增强版本，它管理着一个由同一程序的几个实例或者不同程序的单个实例所共享的数据库。

19.4　对库的一些限制

如前所述，动态链接库模块不接收消息，但是，库模块可调用 GetMessage 和

PeekMessage 。实际上，从消息队列中得到的消息是发给调用库函数的程序的。一般说来，库是代表调用它的程序工作的，这是一条对库所调用的大多数 Windows 函数都适用的规则。

动态链接库可以从库文件或者从调用库的程序文件中装载资源(如图标、串和位图）。装载资源的函数需要实例句柄。如果库使用它自己的实例句柄（初始化期间传给库的），则库能从它自己的文件中获得资源。为了从调用程序的.EXE 文件中得到资源，库函数需要调用该函数的程序的实例句柄。

在库中登录窗口类和创建窗口需要一点技巧。窗口类结构和 CreatWindows 调用都需要实例句柄。尽管在创建窗口类和窗口时可使用库模块的实例句柄，但在库创建窗口时，窗口消息仍发到调用库的程序的消息队列。如果用户必须在库中创建窗口类和窗口，最好的方法可能是使用调用程序的实例句柄。

因为模态对话框的消息是在程序的消息循环之外接收到的，因此用户可以在库中调用 DialogBox 来创建模态对话框。 实例句柄可以是库句柄，并且 DialogBox 的 hwndParent 参数可以为 NULL 。

19.5 不用输入的动态链接

除了在第一次把用户程序装入内存时，由 Windows 执行动态链接外，程序运行时也可以把程序同库模块链接到一起。例如，你通常会这样调用 Rectangle 函数:

```
Rectangle (hdc, xLeft, yTop, xRight, yBotton) ;
```

因为程序和 GDI32.LIB 输入库链接，该库提供了 Rectangle 的地址，因此这种方法有效。

你也可以用更迂回的方法调用 Rectangle 。首先用 typedef 为 Rectangle 定义一个函数类型:

```
typedef BOOL (WINAPI *PFNRECT) (HDC, int, int, int, int);
```

然后定义两个变量:

```
HANDLE hLibrary ;
PFNRECT pfnRectangle ;
```

现在将 hLibrary 设置为库句柄，将 pfnRectangle 设置为 Rectangle 函数的地址：

```
hLibrary = LoadLibrary ("GDI32.DLL") ;
pfnRectandgle = (PFNRECT) GetProcAddress (hLibrary, "Rectangle") ;
```

如果找不到库文件或者发生其它一些错误， LoadLibrary 函数返回 NULL 。现在你可以调用函数然后释放库：

```
pfnRectangle (hdc , xLeft , xLeft , yTop , xRight , ybottom) ;
FreeLibrary (hLibrary) ;
```

尽管此技术并没有为 Rectangle 函数增加太多好处，但它肯定是有用的。如果直到运行

时还不知道库模块的名称，这时就需要使用它。

上面的代码使用了 LoadLibrary 和 FreeLibrary 函数。Windows 为所有的库模块提供“引用计数”，LoadLibrary 使引用计算递增。当 Windows 装载任何使用了库的程序时，引用计数也会递增。FreeLibrary 使引用计数递减，在使用了库的程序实例结束时也是如此。当引用计数为零时，Windows 将从内存中把库删除掉，因为不再需要它了。

19.6 纯资源库

可由 Windows 程序或其它库使用的动态链接库中的任何函数都必须被输出。然而，动态链接库也可以不包含任何输出函数。那么，DDL 包含什么呢？答案是资源。

假设用户正在使用需要几幅位图的 Windows 应用程序进行工作。通常要在程序的资源描述文件中列出资源，并用 LoadBitMap 函数把它们装入内存。但用户可能希望创建若干套位图，每一套均适用于 Windows 所使用的不同显示适配器。将不同套的位图存放到不同文件中可能是明智的，因为只需要在硬盘上保留一套位图。这些文件就是纯资源文件。

图 19-5 说明如何创建包含 9 幅位图的名为 BITLIB.DLL 的纯资源库文件。BITLIB.RC 文件列出了所有独立的位图文件并为每个文件赋予一个序号。为了创建 BITLIB.DLL，需要 9 幅名为 BITMAP1. BMP、BITMAP2. BMP 等等的位图。你可以使用附带的软盘上提供的位图或者在 Windows PAINT 程序中创建这些位图。

BITLIB.MAK

```
#----------------------
# BITLIB.MAK make file
#----------------------

bitlib.dll : bitlib.obj bitlib.res
     $(LINKER) $(DLLFLAGS) -OUT:bitlib.dll bitlib.obj bitlib.res $(GUILIBS)

bitlib.obj : bitlib.c
     $(CC) $(CFLAGS) bitlib.c

bitlib.res : bitlib.rc
     $(RC) $(RCVARS) bitlib.rc
```

BITLIB.C

```
/*--------------------------------------------------------------
   BITLIB.C -- Code entry point for BITLIB dynamic link library
               (c) Charles Petzold, 1996
  --------------------------------------------------------------*/

#include <windows.h>

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvReserved)
     {
     return TRUE ;
     }
```

BITLIB.RC

```
/*---------------------------
   BITLIB.RC resource script
  ---------------------------*/

1 BITMAP bitmap1.bmp
2 BITMAP bitmap2.bmp
3 BITMAP bitmap3.bmp
4 BITMAP bitmap4.bmp
5 BITMAP bitmap5.bmp
6 BITMAP bitmap6.bmp
7 BITMAP bitmap7.bmp
8 BITMAP bitmap8.bmp
9 BITMAP bitmap9.bmp
```

图 19-5 BITLIB库

图 19-6 中所示的 SHOWBIT 程序，从 BITLIB 中读入位图资源，并在客户区的左上角绘制这些位图。你可以通过按一个键来循环显示这些位图。

SHOWBIT.MAK

```
#-----------------------
# SHOWBIT.MAK make file
#-----------------------

showbit.exe : showbit.obj
     $(LINKER) $(GUIFLAGS) -OUT:showbit.exe showbit.obj $(GUILIBS)

showbit.obj : showbit.c
     $(CC) $(CFLAGS) showbit.c
```

SHOWBIT.C

```
/*-----------------------------------------------------------
   SHOWBIT.C -- Shows bitmaps in BITLIB dynamic link library
                (c) Charles Petzold, 1996
  -----------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char  szAppName[] = "ShowBit" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASSEX   wndclass ;

     wndclass.cbSize        = sizeof (wndclass) ;
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
```

```
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wndclass) ;

     hwnd = CreateWindow (szAppName, "Show Bitmaps from BITLIB (Press Key)",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

void DrawBitmap (HDC hdc, int xStart, int yStart, HBITMAP hBitmap)
     {
     BITMAP bm ;
     HDC    hMemDC ;
     POINT  pt ;

     hMemDC = CreateCompatibleDC (hdc) ;
     SelectObject (hMemDC, hBitmap) ;
     GetObject (hBitmap, sizeof (BITMAP), (PSTR) &bm) ;
     pt.x = bm.bmWidth ;
     pt.y = bm.bmHeight ;

     BitBlt (hdc, xStart, yStart, pt.x, pt.y, hMemDC, 0, 0, SRCCOPY) ;

     DeleteDC (hMemDC) ;
     }

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static HINSTANCE  hLibrary ;
     static int        iCurrent = 1 ;
     HBITMAP           hBitmap ;
     HDC               hdc ;
     PAINTSTRUCT       ps ;

     switch (iMsg)
          {
          case WM_CREATE :
```

```
               if ((hLibrary = LoadLibrary ("BITLIB.DLL")) == NULL)
                    DestroyWindow (hwnd) ;

               return 0 ;

          case WM_CHAR :
               if (hLibrary)
                    {
                    iCurrent ++ ;
                    InvalidateRect (hwnd, NULL, TRUE) ;
                    }
               return 0 ;

          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               if (hLibrary)
                    {
                    if (NULL == (hBitmap = LoadBitmap (hLibrary,
                                        MAKEINTRESOURCE (iCurrent))))
                         {
                         iCurrent = 1 ;
                         hBitmap = LoadBitmap (hLibrary,
                                        MAKEINTRESOURCE (iCurrent)) ;
                         }

                    if (hBitmap)
                         {
                         DrawBitmap (hdc, 0, 0, hBitmap) ;
                         DeleteObject (hBitmap) ;
                         }
                    }

               EndPaint (hwnd, &ps) ;
               return 0 ;

          case WM_DESTROY :
               if (hLibrary)
                    FreeLibrary (hLibrary) ;

               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

图 19-6 SHOWBIT程序

在处理 WM_CREATE 消息时，SHOWBIT 得到 BITLIB.DLL 的句柄：

```
if ((hLibrary = LoadLibrary ("BITLIB.DLL")) == NULL)
   DestroyWindow(hwnd) ;
```

如果 BITLIB.DLL 不在当前目录，那么 SHOWBIT 将按前面所述的方法进行搜索。如果无法找到此库，那么 Windows 将显示一个消息框报告这个问题。用户按下 OK 时，LoadLibrary

返回一个 NULL 值，并终止 SHOWBIT 的运行。

SHOWBIT 通过库句柄和位图号来调用 LoadBitMap ，从而得到一个位图句柄：

```
hBitMap = LoadBitMap (hLibrary, MAKEINTERSOURCE (iCurrent)) ;
```

如果号码 iCurrent 对应的位图无效或者没有足够的内存装载位图，则返回一个错误。

在处理 WM_DESTORY 消息时， SHOWBIT 释放位图库:

```
FreeLibrary (hLibrary) ;
```

当 SHOWBIT 的最后一个实例终止时， BITLIB.DLL 的引用计数变为 0 ，并且释放所占用的内存。

第20章 何谓 OLE

OLE 是用于构造可连接组件（component）软件的一组标准。其中一种 OLE 标准是组件对象模型（COM）规范，这是为组件之间的二进制连接所定义的。OLE 的另一个定义成分是一组动态链接库，这些库是 Window 95 和 Windows NT 的组成部分。Microsoft 为第三方合作伙伴——包括 Digital、Software AG 和 Bristol Technologies——做了安排，将一些技术从这些库中移植到其它的操作系统平台。不过，由 OLE 提供的一组服务并非是静态的。就和 Microsoft 不断地在增强和扩展 Windows 操作系统一样，它还在不断地升级 OLE，以满足范围越来越广的应用集成的需要。例如，在写作这本书时，Network OLE 被包含进了 Windows NT 版本 4.0 中作为它的组成部分；它将允许跨网络的组件连接。

OLE 允许在软件模块之间的一定程度集成——这在以前是需要具有专门知识才能做到的，从而迈出了创造一个可交换软件组件世界的第一步。在这个世界里，用户可以将来自一个厂商的字处理编辑窗口同来自另一个厂商的拼写检查器组合到由第三个厂商提供的打印预览组件中。

目前，只建立了几类标准的 OLE 连接。不过，OLE 的体系结构不仅允许更多的连接类，而且建立更新的连接族已是不可避免的。在 Windows 95 之前，OLE 的四类最常用连接是复合文档支持、OLE 动画、OLE 控件以及扩展消息应用编程接口（扩展 MAPI）。Windows 95 引入了一族新的 OLE 组件，叫做“外壳扩展（shell extensions）”，用于构造应用软件和 Windows 95 桌面之间的紧密集成。另一个由 Windows 95 引入的有关 OLE 的组件是丰富编辑控制（rich edit control），这是一个提供简单的复合文档容器的对话框。

一个复合文档容器创建复合文档，该文档可以包含来自许多不同的互不相关的应用程序的数据。容器和“对象服务器”通过通讯来协调在服务器和复合文档之间双向的数据移动。一个 Excel 电子表格的一部分嵌入到一个 Microsoft Word for Windows 的文件中，这就是复合文档的一个例子。在这个例子中，Excel 电子表格数据是一个嵌入对象，而 Word for Windows 提供复合文档容器。然而，这个例子并不能最好地说明 OLE，因为两个来自同一公司的产品紧密合作，容易让人觉得没什么了不起。OLE 的复合文档支持确实是广泛的，这可以通过许多厂商——例如，Adobe PageMaker 或者 Micrografx Designer——提供与 Microsoft Word 同样的复合文档支持来说明。每一个容器应用程序都可以容纳 Excel 电子表格对象、Corel Draw 绘图对象、Visio 图表以及来自任何一个 OLE 兼容对象服务器的数据对象。

动画提供了定义一组宏原语的机制。宏原语包括方法（函数调用的另一种叫法）和属性（即，可以加以读写的数据元素，或者只能被读或者只能被写的数据元素）。术语“动画对象”是指提供宏原语的 OLE 组件。“动画控件”操纵动画对象的方法和属性。通过 OLE 定义的动画标准，诸如 Microsoft 的 Visual Basic、Borland 的 Delphi 或者 PowerSoft

的 Power Builder 等编程环境允许你创建动画控件，从而对分散在专门的应用上的工作提供集中控制。

OLE 控件是第三种标准 OLE 组件；与对话框控制类似，它们本质上是具有特殊目的的子窗口。然而，OLE 控件并不处在对话框中，而是处在 OLE 控件容器中。OLE 控件是自主的对象，并可以输出它们自己的宏原语。与复合文档对象相类似，OLE 控件可以在它们的容器应用所创建和维护的文件中保存状态信息。OLE 控件识别环境属性，比如背景色和当前字体等，从而可以使它们在视觉上融入它们的容器中。OLE 控件能够向控制容器发送事件通知，在这一点上，它们类似于 Visual Basic 控件，这些控件发送事件以触发响应。事实上，Microsoft 公开表示 32 位环境将不支持 Visual Basic 控件（VBX），并鼓励所有 VBX 开发人员将他们的 Win16 VBX 升级为 Win32 的 OLE 控件。

所有标准 OLE 连接的共性在于它们和 OLE 组件对象模型（COM）紧密结合。COM 是面向对象的接口（即组件之间的接触点）定义规范，是构筑于其上的所有 OLE 功能特性的基础。

在我进一步深入之前，先提醒你一点，组成 OLE 的服务集合是非常大的——并且还在增大！由于本章没有足够的空间来涉及 OLE 的所有方面，因此重点将放在组件对象模型的基本原理上。在讨论中，你将看到两个 OLE 服务器（一个是私有组件，另一个是公共组件）和两个 OLE 客户程序。有关 OLE 编程问题的更详细讨论，请参考由 Kraig Brockschmidt(Microsoft Press, 1995)编著的 Inside OLE 。

20.1 OLE 基础

使用 OLE 组件支持要求遵循一些基本规则。

20.1.1 连接到 OLE 库

在可以执行任何 OLE 操作之前，Win32 进程必须连接到 OLE 库。进程中的任何模块——无论是程序还是 DLL——都可进行此连接。由于进行多次连接是有益的，并且由于一个模块的开发人员无法保证其它模块是否进行了连接，所以每个依赖于 OLE 库的模块都应该尝试进行一次连接。

连接 OLE 库最常用的方法是调用 OleInitialize：

```
HRESULT hr = OleInitialize (NULL) ;
if (FAILED (hr))
   {
   //错误处理
   }
```

OleInitialize 唯一的参数必须是 NULL（在 16 位 OLE 中，其它值也是有效的，但是在 32 位 OLE 中，只有 NULL 才是有效的）。这个函数为当前进程建立 OLE 服务，然后调用另一个函数，CoInitialize，来初始化组件对象模型库。这个库提供组件创建和清除这类低层

服务。

在某些情况下，你可能希望跳过 OleInitialize 而直接调用 CoInitialize 。例如，在建立一个只要求最基本的 COM 服务的自定义 OLE 组件时，可以通过调用 CoInitialize 来连接到 OLE 。主要的好处是可以占用更少的存储，因为只有 OLE 的多个 DLL 中的一部分被真正装入存储中。对 CoInitialize 的调用和对 OleInitialize 的调用相同：

```
HRESULT hr = CoInitialize (NULL) ;
if (FAILED (hr))
{
// 错误处理
}
```

如果在建立连接时没有返回失败代码，那么每个创建 OLE 连接的 OLE 组件都应该在终止之前自己断开连接。 OleUninitialize 是对应于 OleInitialize 的断开连接的函数，而 CoUnintialize 是对应于 CoInitialize 的断开连接的函数。可以想象， OleUninitialize 调用 CoUninitialize ，因此一个组件只能调用和它调用的连接函数相对应的函数来断开连接。

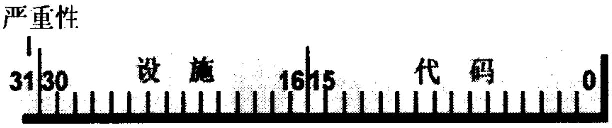
两个初始化函数的返回值类型都是 HRESULT ，它告诉调用者调用成功或失败。前面的代码段显示了使用 FAILED 宏捕获失败的方法。另一个相反的宏， SUCCEEDED ，在成功时返回 TRUE 。几乎所有的 OLE 函数都返回类型为 HRESULT 的值。下面一些准则将帮助你正确地解释返回值的含义。

20.1.2 解释结果代码

贯穿 OLE 的一个特征在于对标准返回类型 HRESULT 的使用，几乎所有的 OLE 库调用和 OLE 接口调用都使用这种返回值。 OLE 的 include 文件定义了这种类型：

```
typedef LONG HRESULT ;
```

虽然被称作“结果句柄”，但它并不是通常那种指向某个对象的句柄，而是由位域组成的一个 32 位值，如图 20-1 所示。



严重性：	（1 位）	严重性域
		0　成功。函数调用成功。
		1　错误。函数由于某种错误情况而失败。
设施：		（15 位）表示此返回值属于哪一组状态码。Microsoft 保留定义此域的权利。
代码：		（16 位）描述实际上发生了什么，错误或其它情况。

图 20-1　HRESULT的域

HRESULT 中最重要的域是严重性域，它表示成功或失败。 SUCCEEDED 和 FAILED

宏完全基于这个域来确定成功还是失败。

设施（Facility）域标识出引发错误的组件。（请注意，为了避免和 Microsoft 的设施码冲突，不要创建你自己的设施码。）目前，如下面的表所示，只定义了少数设施码：

设施名	设施值	描述
FACILITY_NULL	0	用于广泛可用的公共状态码，这些状态码没有特定的组。（例如，S_OK 属此设施。）
FACILITY_RPC	1	来自底层实现中的远程过程调用的结果。
FACILITY_DISPATCH	2	和 Idispatch 接口有关的结果。
FACILITY_STORAGE	3	和存储相关的结果。小于 256 的结果码含义和 MS_DOS 错误代码的含义相同。
FACILITY_ITF	4	最常用的设施。其含义和接口有关。
FACILITY_WIN32	7	起源于 Win32 的 GetLastError 函数的结果。
FACILITY_WINDOWS	8	来自 Microsoft 定义的接口的结果。
FACILITY_CONTROL	10	和 OLE 控件有关的结果。

有关细节由代码字段的 0 到 15 位给出。对于失败的情况，这个字段包含错误代码。即使对于成功的情况，我们马上将看到，有时也有一些细微的差别值得注意。

在 HRESULT 代码中位的排列可能和实现有关。如果要确定成功或者失败，你应该只根据 FAILED 和 SUCCEEDED 宏。如果要获取一个错误代码的设施，那就使用 HRESULT_FACILITY 宏。而对于错误代码，使用 HRESULT_CODE 宏。这将保证你的 OLE 代码更容易移植，例如，从 Windows 95 移植到 Apple Macintosh 、 UNIX 、 VMS 或者到任何一种 OLE 可以移植到的平台上。

作为 HRESULT 值的一个例子，下面的表总结了 CoInitialize 和 OleInitialize 的返回值：

返回值	二进制值	描述
S_OK	0x00000000L	表示库被成功地初始化
S_FALSE	0x00000001L	表示库已经被初始化过了。
E_OUTOFMEMORY	0x8007000E	由于存储用完导致的失败
E_INVALIDARG	0x80070057	由于非法参数导致的失败
E_UNEXPECTED	0x8000FFFF	由于预料之外的错误而导致的失败

当一个模块试图初始化 OLE 库时， S_OK 和 S_FALSE 的返回值都表示成功的结果。进程中第一个成功的初始化调用返回 S_OK 。接下来的调用不影响已经被初始化的库，而只是反映在 S_FALSE 返回值中。“ S_”前缀代表“成功”，这是对这两个结果代码的精确表示，因为二者都意味着 OLE 可以使用了。当然，在某些情况下，模块可以总是对某些特定的值进行检查，比如，当一个调试工具需要成为初始化 OLE 库的第一个模块时。

OLE 文档中列出了大多数函数的一些结果代码，但是一个函数失败时的结果代码可能（而且经常）不在文档中为该函数列出的代码里。识别出 HRESULT 代码的含义并不容易，

它涉及对 OLE 头文件的搜索。一种更简单的方法是调用 FormatMessage，这是一个 Win32 例程，用于将消息代码（或者在这种情况下，一个 OLE 结果句柄）翻译为一个可读的字符串。下面是 FormatMessage 的一个包装函数，它接受一个 OLE 或者 Win32 错误代码并提供该代码的文本描述：

```
// DisplayErrorString: Fetch explanation for
//   Win32 and OLE error codes
void DisplayErrorString (DWORD dwError, BOOL bDebugger)
     {
     .CHAR achBuffer[120] ;
     FormatMessage (FORMAT_MESSAGE_FROM_SYSTEM, NULL,
                    dwError,
                    LANG_SYSTEM_DEFAULT,
                    achBuffer, 120,
                    NULL) ;

     if (bDebugger)
          {
          OutputDebugString (achBuffer) ;
          }
     else
          {
          MessageBox (NULL, achBuffer,    escribeError", MB_OK) ;
          }
     }
```

这个函数要求两个参数：dwError 是由一个 OLE 函数返回的 HRESULT 值，而 bDebugger 是一个布尔值，指示在向哪里输出。如果 bDebugger 是 TRUE，那么输出进入调试器的输出窗口（如果调试器没有运行，这类输出将消失）或者进入一个消息框。

凑巧的是，OLE 的 HRESULT 值被放入了 Win32 错误代码中，你仔细阅读 Win32 的头文件 WINERROR.H 就可以看到。OLE 和 Win32 处理错误代码之间的区别在于：大多数 OLE 函数直接返回一个 HRESULT 值，而 Win32 函数并不直接返回一个错误代码。对于 Win32 函数，这些错误代码是通过调用 GetLastError 获得的。这个函数返回最近一个在当前线程中失败的函数的当前错误代码。有些 Win32 例程会清除错误代码——在调用可能会失败的其它 Win32 例程之前将错误代码置为一种已知状态——因此在调用其它 Win32 函数之前首先对这个代码进行检查是必要的。

迄今为止，我们接触了 OLE 编程的两个基本点：首先，要将 OLE 作为一个组件使用，你必须首先和 OLE 库连接；第二，对 HRESULT 返回代码的适当解释，区别出成功和不成功的连接。和 OLE 库连接之后，一个模块有两种选择：或者连接到某个 OLE 接口（即，成为某个 OLE 接口的客户），或者使某个 OLE 接口可以为其它对象使用（即，成为某个 OLE 接口的服务器）。大多数 OLE 有关的程序都同时担任这两种角色。为了帮助你决定应该首先考虑选择哪一个，下一节将介绍 OLE 接口的基本原理。OLE 接口是遵循“组件对象模型（COM）”规范而创建的，因此有时也被称为 COM 接口。

20.1.3 组件对象模型（COM）接口

组件对象模型（COM）是构造有关 OLE 组件的规范，它定义了接口提供者和接口使用者之间的基本关系。所有的OLE接口——无论是针对复合文档、OLE动画、OLE控件、Windows 95 外壳扩展或者是扩展 MAPI——都是按照 COM 规范建立的。

COM 接口为一个问题提供了单一的解决方案，以前曾经针对这个问题提出过许多的解决方案。这个问题就是：将在不同进程中或者在不同计算机上运行的软件组件连接在一起。简单说来，其中的难点在于如何将由一种地址空间所限定的，一种网络所限定的，或者根本没有界限将其分开的组件有效地连接起来（写这本书时，Network OLE 还没有公布，但是可望在这本书发行后不久公布）。OLE 接口既提供进程间和机器间的接口，同时还要有效地处理同一地址空间中运行的组件之间的连接。对这三种连接情况的单一解决方案意味着开发人员可以完全忽略仅仅由于组件位置所引起的区别，而将重点放在手头的任务上。这类似于电话的使用方式：不论是在同一建筑物中的不同房间，还是在同一大陆的两边，或者在同一行星的不同半球，人们都使用同一机制——电话机——来进行通讯。

和所有好的设计一样，接口将多个事物接合在一起。接口既是一种服务的合同，又是服务提供者和服务使用者之间的边界连接，也是支持进程外连接（即，包括进程间和机器间）的一种机制。将这些方面捆在一起的是由所有接口共享的一组核心函数。这里的每一方面都值得进行更仔细地研究。

20.1.3.1 接口是一种服务合同

一个接口是由一个组件（“服务器”）提供并由另一个组件（“客户”）使用的服务合同。这样，和一个接口定义有关的唯一代码由一组函数原型——函数返回值、函数名、和函数参数——组成。合同规定提供哪些服务，而不是如何提供这些服务。实现的细节被隐藏——即封装——在服务提供者中。

例如，下面是 IMalloc —— OLE 的存储分配接口——的函数原型：

函数原型	描述
void *Alloc (ULONG cb) ;	分配 cb 个字节。
void *Realloc (void *pv, ULONG cb) ;	将 pv 存储的大小改变为 cb 个字节。
void Free (void *pv) ;	释放由 pv 指向的存储。
ULONG GetSize (void *pv) ;	查询存储大小。
int DidAlloc (void *pv) ;	是否由这个分配器分配的？
void HeapMinimize (void) ;	收集存储碎片。

由这样的一个分配器提供的服务——即“什么”——对具有动态存储分配经验的程序员来说是浅显的，但是它的实现——即，“如何”满足要求——并没有说明。在 Windows 95 上的一种 IMalloc 实现可以使用任何可用的 Win32 技术，包括针对大对象的页面分配器（VirtualAlloc）、针对小对象的堆分配器（HeapAlloc）以及与 Win16 和 Win32 API 都

兼容的一个 Win16 兼容分配器（GlobalAlloc 和 LocalAlloc）。一个实现者甚至可以调用 C 的运行分配函数来创建 Win32 API 无关的一种可移植 IMalloc。

关于 IMalloc，值得一提的是接口的命名标准。遵循匈牙利记法的命名风格，接口名有一个“I”前缀：例如，Istorage、IOleObject 和 IDataObject。这类似于对 Microsoft MFC 库类名使用“C”前缀 (CWinApp、CFrameWnd)或者对 Borland OWL 的类名使用“T”前缀（Tapplication、TWindow）。

虽然如何实现一个接口的细节隐藏不可见，但客户和服务器之间的二进制接触点却清楚地定义了出来。

20.1.3.2 接口定义一个二进制接触点

在其核心，一个接口的二进制实现是一个函数指针的数组。一个象 IMalloc 这样的接口的服务器实现各个函数并构造一个函数指针数组，客户希望通过这样的数组来访问服务器函数。

服务器提供给客户的特定连接是指向一个函数指针数组的指针。在 C 中，数组语法提供一种构造函数指针数组的方法。一个 IMalloc 服务器可以按如下所示来分配函数指针数组：

```
// 定义通用函数指针类型
typedef (* PFNINTERFACE) ( ) ;

// 分配一个函数指针数组
PFNINTERFACE allocator[6] = { Alloc, Realloc, Free,
                              GetSize, DidAlloc,
                              HeapMinimize } ;
```

然后可以将关键的连接点——指向函数指针数组的指针——返回给调用下列函数的任何一个客户：

```
PFNINTERFACE * FetchAllocator ( )
    {
    return & allocator ;
    }
```

数组语法的问题在于对 IMalloc 函数的调用很笨拙并且容易出错。返回类型必须被强制，不进行参数检查，并且数组索引代替了函数的名字。例如，对 Alloc 的调用要求 20 个字节，如下所示：

```
PFNINTERFACE *pAlloc = FetchAllocator ( ) ;
void * pData = (void*)pAlloc[0] (20) ;
```

在禁止参数类型检查的情况下，如下所示的调用不会产生任何警告或者错误信息：

```
// 错误的参数类型！！
pData = (void*)pMalloc[0] ("20 bytes") ;
```

```
// 错误的参数个数！！
pData = (void*)pMalloc[0] (20, 30, 40, 50, 60) ;
```

一种更好的方法用到一个包含函数指针的结构。采用这种方法需要类似如下的定义：

```
// Specifically-typed function pointer types
typedef void * (* PFNALLOC) (ULONG) ;
typedef void * (* PFNREALLOC) (void *, ULONG) ;
typedef void   (* PFNFREE) (void *) ;
typedef ULONG  (* PFNGETSIZE) (void *) ;
typedef int    (* PFNDIDALLOC) (void *) ;
typedef void   (* PFNHEAPMINIMIZE) (void) ;

// Define structure containing function pointers
typedef struct tagALLOCATOR
     {
     PFNALLOC          Alloc ;
     PFNREALLOC        Realloc ;
     PFNFREE           Free ;
     PFNGETSIZE        GetSize ;
     PFNDIDALLOC       DidAlloc ;
     PFNHEAPMINIMIZE   HeapMinimize ;
     } ALLOCATOR ;

ALLOCATOR imalloc = { Alloc, Realloc, Free,
                      GetSize, DidAlloc,
                      HeapMinimize } ;

ALLOCATOR *FetchAllocator ()
     {
     return &imalloc ;
     }
```

虽然没有显式地使用数组语法，但是二进制映象的实际排列类似于上一例中的函数数组。

在这里，结构的使用——虽然从某种程度上来说可能它的定义比较复杂——使调用很容易而且很自然。尤其是，不要求对返回类型进行强制转换，可以使用函数名来进行函数调用，并且编译器对参数类型的检查完全可行。下面是一个分配 20 个字节的调用：

```
ALLOCATOR *pAlloc = FetchAllocator ( ) ;
void * pData = pAlloc->Alloc (20) ;
```

最好的方法—— OLE 接口采用的方法——是使用在定义 C++类时自动产生的函数指针数组。为全部“虚拟”函数——使用 virtual 关键字为前缀的那些函数——创建一个函数指针数组。在 C++类定义中函数原型的出现支持对参数的检查，而无需针对函数指针的没完没了的 typedef：

```
class IMalloc
     {
     virtual void  *Alloc (ULONG cb) ;
```

```
virtual void  *Realloc (void *pv, ULONG cb) ;
virtual void   Free (void *pv) ;
virtual ULONG  GetSize (void *pv) ;
virtual int    DidAlloc (void *pv) ;
virtual void   HeapMinimize (void) ;
} ;
```

C++类的使用会带来一些小小的复杂性。尤其是，当从 C 调用这样的函数时， C++对象要求额外的一层间接性，即要求对函数表格的引用。最终创建的是一个 IMalloc **而不是一个简单的 IMalloc *指针。另一个复杂性是组件对象的指针必须作为第一个参数传递，以模拟所有 C++对象需要的一个 this 指针的隐含传递。下面说明如何在 C 中调用 IMalloc 接口函数：

```
IMalloc *pMalloc = FetchMalloc ( ) ;
void *pData = pMalloc->lpVtbl->Alloc (pMalloc, 20) ;
```

如果不考虑这些额外的复杂性，一个接口的二进制定义的核心——函数表数组——是相同的。

OLE 接口的一个重要方面是对进程间和机器间的支持，在这里对 OLE 的介绍中，我们将统称为“远程连接”。

20.1.3.3 接口是一种远程连接

不论一个客户/服务器的连接是进程内的或者是跨进程或网络界限的，对客户来说，二进制接口——函数数组——所处的地位都是一样的。当服务器和其客户是在同一进程中运行时，函数数组将客户的调用与服务器中的相应函数结合在一起。当调用进程外的服务器时，客户仍然看到一个函数数组。存在两种进程外的服务器：在同一台机器上的不同地址空间中运行的本地服务器和在不同机器上运行的远程服务器。对调用者来说，本地服务和远程服务之间的区别是透明的。服务器提供必要的支持技术，以保证它的接口能够提供一种无缝连接——无论是跨过进程界限还是跨过网络界限，而不需要客户知道或者担心“真正”的工作是在哪里完成的。

OLE 库对远程运行的需求提供了许多支持。在客户进程的地址空间中，构造一个函数数组并将它连接到一组代理函数(proxy functions)。代理函数在逻辑上等价于在电话系统中的呼叫转发，它将参数打包，这个包将被传送到服务器地址空间中“真正”的接口。在服务器的地址空间中，有一组桩函数(stub functions)将这个包解开，并使用传过来的参数调用实际的服务器函数。

由代理和桩函数进行的打包、传送和解包工作被称为整理（ marshaling ）。 OLE 使用一种内部的跨进程和跨网络远程过程调用（ RPC ）机制，对所有 OLE 定义的接口提供缺省的整理支持。如果你定义了一种用户接口——即不属于 OLE 标准的新接口——那么你的接口不会继承这些远程支持。

为了使一个用户接口支持远程连接， OLE 为你提供两种选择。第一种选择是编写一个接口定义语言（ IDL ）文件，描述你的接口并将该文件传给 Microsoft IDL 编译器。这个编译器将产生代理函数和桩函数代码，你可以将这些代码编译成一个 DLL 。第二种选择是由

你自己来控制整理过程的所有方面，这允许你使用任何传输技术——诸如共享存储、Windows 消息、有名管道、RPC、Morse 代码，等等。这种选择被称为“用户整理”，意味着 OLE 本身将不涉及跨进程和跨网络的调用。

一个 OLE 接口函数表包含两组函数：接口通用函数和接口特定函数。前面 IMalloc 接口的例子只显示了特定于存储分配任务的函数，而省略了三个通用函数，这三个函数占据所有 OLE 接口函数数组的前三个位置，它们将在下一节中介绍。

20.1.3.4 每个OLE接口都有这三个函数

所有的 OLE 接口都有一组通用的接口函数，它们支持 OLE 组件所需要做的一些基本工作：QueryInterface、AddRef 和 Release。这三个函数一起组成它们自己的接口，称为 IUnknown:

```
class IUnknown
    {
    virtual HRESULT QueryInterface (REFIID iid, void **ppv) ;
    virtual ULONG   AddRef ( ) ;
    virtual ULONG   Release ( ) ;
    }
```

命名为“Unknown”的接口所遵循的原理反映了一个事实：接口是通过指针来访问的，从而有助于创建一个通用指针。和 C/C++的 void *一样，一个“通用指针”提供可由任何接口完成的最小一组已知信息。具体地说，你将遇到返回不同类型接口指针的 OLE 函数，这一类函数经常被原型化为返回一个 IUnknown *。

出于简单性的考虑，前面介绍过的 IMalloc 接口不包括这三个接口函数。一种更精确的 IMalloc 定义引用 IUnknown 作为基类，从中继承它所有的成员：

```
class IMalloc : public IUnknown
    {
    virtual void *Alloc (ULONG cb) ;
    virtual void *Realloc (void * pv, ULONG cb) ;
    virtual void  Free (void *pv) ;
    virtual ULONG GetSize (void *pv) ;
    virtual int  DidAlloc (void *pv) ;
    virtual void HeapMinimize (void) ;
}
```

给 C 程序员的提示

通过 C++类的继承技术，在 IMalloc 的定义中附加“:public Iunknown”后缀将所有 IUnknown 的成员加到 IMalloc 类中。九个成员函数组成 IMalloc 的虚函数表，并且这九个函数定义了对这个接口的二进制连接。

除了这六个存储管理函数外，IMalloc 接口还提供三个和存储管理无关的函数。你将看到，这三个函数为 OLE 组件提供了重要协作机制。与任何其它接口一样，IUnknown 表

示一种服务协议。但是因为这个接口在每个其它的接口中都会出现——用C++的术语来说，IUnknown 是其它每一个 OLE 接口的基接口——所以 IUnknown 定义了每一种 OLE 接口必须提供的服务。理解这一约定的含义是很重要的，不管你是在实现一个客户程序或者服务器程序。

20.1.4 IUnknown 服务

虽然出现为单个的接口，这三个 IUnknown 函数实际上代表了 OLE 组件服务而不是接口特定的服务。两个基本的服务是：组件生命周期的管理和在单个组件中对多个接口的访问（注意：术语“ OLE 组件”等价于在 OLE 规范中使用的术语“ OLE 对象”）。

20.1.4.1 管理“组件生命周期”

组件生命周期是指当不再需要组件时，要对它们进行“关闭”操作。如果没有这一关键步骤，那么即使最富有存储的系统都会最终将存储用完。基本的技术是很直观的：每一个 OLE 组件（即 OLE 对象）维护一个引用计数，当组件被创建时从 1 开始记数。对 QueryInterface 和 AddRef 的调用将引用计数加 1，而对 Release 的调用将引用计数减 1 。当计数变为 0 时，一个组件的生命周期结束，释放其拥有的所有系统资源。

在最简单的情况下，对 Release 的一个调用就可以切断客户和接口之间的连接。每种接口都支持这个函数，这意味着你不必区分接口特定或者组件特定的终止函数；一个函数可以适用于所有接口。下面的代码演示了连接到 OLE 任务分配器（ IMalloc 的一种实现），使用它，并切断与它的连接的过程。 OLE 组件使用这个分配器为一个组件分配存储，并由另一个组件来释放所分配的存储：

```
char    *pData ;
HRESULT  hr ;
IMalloc *pMalloc ;

// Connect to task allocator
hr = CoGetMalloc (1, &pMalloc) ;
if (SUCCEEDED (hr))
     {
     // Allocate memory
     pData = pMalloc->Alloc (cbBytes) ;
     if (pData)
          {
          lstrcpy (pData, pSource) ;
          }
     // Disconnect from task allocator
     pMalloc->Release () ;
     }
```

调用 IMalloc::Release 并不会释放被分配的存储，但是它会将分配器组件上的一个引用计数减 1 。要释放存储，可以调用 IMalloc 的 Free 成员函数：

```
hr = CoGetMalloc (1, &pMalloc) ;
if (SUCCEEDED (hr))
    {
    pMalloc->Free (pData) ;
    pMalloc->Release ( ) ;
    }
```

为了简化对任务分配器的访问，在 OLE 库中增加了两个更简单的包装函数：CoTaskMemAlloc 和 CoTaskMemFree 。分配存储时使用：

```
pData = CoTaskMemAlloc (cbBytes) ;
```

释放存储时使用：

```
CoTaskMemFree (pData) ;
```

对组件生命周期更复杂的管理发生在当一个客户的许多不同部分都依赖于接口服务时。这一复杂性的一部分是由于这样一个简单的事实：程序员一般都不习惯使用引用计数来管理对象的生命周期，虽然他们有两种类似的管理经验：在程序的世界中，管理 GDI 对象；在人类世界中，通过电话会议进行交流。

一个 GDI 绘图对象——一支画笔、一把画刷、一种字体等等——必须明确地被创建，明确地被删除（即使 Windows 95 和 Windows NT 会在一个进程终止时清除这一类的对象，但是由你自己来进行清除仍然是一种好的编程习惯）。也就是说，对 GDI 中创建一支绿色画笔的每个调用：

```
hpenGreen = CreatePen (PS_SOLID, 1, RGB (0, 255,0)) ;
```

都要求一个相应的调用来删除它：

```
DeleteObject (hpenGreen) ;
```

从引用计数的观点来看，GDI 对象永远不会有一个大于 1 的引用计数值。

这种方法的缺点是两个程序——比如 PageMaker 和 Visio ——可能分别创建一支同样的绿色画笔，或者在每个程序中的不同子系统——用户接口、打印驱动、位图支持和元文件支持——可能分别创建一支自己的绿色画笔。这样可能会创建 10 支同样的画笔，导致系统资源的浪费。

然而，如果 GDI 使用一种基于引用计数的技术来跟踪系统范围内画笔的创建，那么第二个及其后面的创建绿色画笔的请求都将导致在第一支绿色画笔的引用计数上加 1 。好处是减少了系统的负荷，因为一支绿色画笔就完成了 10 支绿色画笔所做的工作。如果 GDI 画笔是作为 OLE 组件实现的，那么在第一次以后的每支画笔的创建都将归到对绿色画笔组件的这样一个调用：

```
pPenGreen->AddRef ( ) ;
```

每次调用会将画笔引用计数加 1 。每次对 DeleteObject 的调用将通过下面的一个调用将引

用计数减 1 ：

```
pPenGreen->Release ( ) ;
```

直到引用计数为 0 时，意味着绿色画笔可以被清除了。对组件生命周期使用基于引用计数的控制，使得一个组件的每个客户和组件的连接有一个明确的开始和结束（附带说一句，值得指出，在 Windows 95 下的 GDI 使用了引用计数的技术来最小化冗余对象，虽然它并没有基于 OLE 接口）。

引用计数代表与一个组件的一种独立的连接。这与你处理你公司和你公司最佳客户代表之间的一次电话会谈一样。可能你作为你公司的技术代表，需要回答一个问题。在这种情况下，只有一个引用计数 1 。

如果交谈转变为测试方法方面的话题，那么你可能会要求测试经理——在隔壁办公室的同事——加入会谈。当它拿起电话说“你好”的时候，就像是对 AddRef 的一次调用——会议电话的引用计数增加为 2 。当话题转到文档方面时，你可能要求文档管理员拿起电话加入电话会谈之中。当她说“你好”时，引用计数增加为 3 。你的客户知道他的软件供应商的三个雇员在和他通话。只有在最尴尬的情况下客户代表才可能会突然挂断电话，这将类似于一个 OLE 服务器的崩溃。在某些情况下，你的客户对你们的回答感到满意，并且他希望和你的销售经理谈一谈有关订单的事。当销售经理加入这次电话会议时，他的“你好”将电话会议的引用计数增加到 4 。

当你、测试经理和文档管理员得以插话时，每个人都说“再见”然后将电话挂断。每个再见都类似于对 Release 的一次调用。不过，电话会议继续，类似于一个组件还在存储中存在。最后，当销售经理谈完了这个大订单之后，他对客户说“再见”并挂断电话。电话会议的引用计数此时为 0 ，现在你的客户可以挂断电话了。

虽然概念上很简单，但是保证引用计数的正确需要时间和精力。在正确的时间退出要求接口客户正确地计算所有的连接，不论是显式的还是隐含的连接。当一个接口客户没能够正确地管理组件的生命周期时，可能会产生奇怪的结果：没有切断一个连接将使组件逗留在存储中；过早切断连接又可能使某些连接方组处于不可预料的状态。

管理组件生命周期只是由 IUnknown 成员函数执行的一种任务而已。另一种重要的任务是在单个组件中支持多个接口，这是下一个话题。

20.1.4.2 多个接口支持

OLE 的组件对象模型结构允许单个组件在二进制层提供多个接口，即提供多个函数指针数组。多个接口的主要优点是提供比单个接口更大范围的服务能力。接口其实就是服务合同，因此具有多个接口的能力也就是提供多种服务的能力。获取一个组件接口的指针的基本方法是使用 QueryInterface 函数。

由于 QueryInterface 本身也是一个接口函数，因此不能调用它来获取一个组件中的第一个接口指针。但是，它可以用来获取一个组件中的第二个以及以后的接口指针。第一个接口指针几乎总是由一些 C 语言的可调用（非接口式）函数返回的。在前面的一个例子中，CoGetMalloc 获取 OLE 库的任务分配器的一个 IMalloc 指针：

```
HRESULT hr ;
```

```
Imalloc *pMalloc ;

// 和任务分配器连接
hr = CoGetMalloc (1, &pMalloc) ;
```

当获得了第一个接口指针之后，以后的接口指针通过调用任何一个现存接口的 QueryInterface 成员来进行访问。QueryInterface 定义如下：

```
HRESULT QueryInterface (REFIID iid, void **ppv) ;
```

- iid 是一个接口标识符（IID）的指针。
- ppv 指向函数返回值的存储位置。对于所支持的接口，返回值将是所要求类型的一个接口指针。对于不支持的接口请求将返回一个 NULL 值。

如果你浏览 Win32 包含文件，将看到一个 IID 被定义为一个 GUID（全局唯一标识符），而 GUID 本身是在 Win32 头文件 WTYPES.H 中定义的，如下所示：

```
typedef struct _GUID
    {
    DWORD Data1 ;
    WORD  Data2 ;
    WORD  Data3 ;
    BYTE   Data4[8] ;
    } GUID ;
```

这是用来唯一地标识接口的一个 16 字节的值。

如果一个 OLE 组件是一幢建筑物，那么 IID 将是你希望呼叫的部门分机号。在这样的一个世界中，对 QueryInterface 的调用是请求你已经在通话的部门将电话转给同一组织（组件）中的一个新的部门。接口的名字就像部门的名字——会计、人事资源、开发、测试等等。下面是"电话簿"的一小部分，其中列出了在 OLE 组件中遇到的部门的标准分机：

接口名	IID
Iunknown	{00000000-0000-0000-C000-000000000046}
IClassFactory	{00000001-0000-0000-C000-000000000046}
Imalloc	{00000002-0000-0000-C000-000000000046}
IMarshal	{00000003-0000-0000-C000-000000000046}

重要的是记住，可读的名字（"Iunknown"）只是为了方便你，但是你的 OLE 客户和服务器使用二进制数来区分不同的接口。幸运的是，你不必处理这么一长串数字，因为所有的接口 ID 都有符号化的常数对应。符号名在接口名字前增加一个"IID_"前缀，这样 IID_IUnknown 指的是 IUnknown 接口的二进制接口。例如，拥有一个组件指针的客户可以进行如下所示的调用来获取一个 IMarshal 接口的指针：

```
LPMARSHAL pMarshal ;
HRESULT hr = pMalloc->QueryInterface (IID_IMarshal, &pMarshal) ;
```

记住这个调用首先是一个查询："你是否支持 IMarshal 接口？"回答是"是"（HRESULT

为 S_OK）或者“不”（HRESULT 为 E_NOINTERFACE）。原因是存在几百种接口，并不是所有的组件都支持所有的接口。

如果所请求的接口被支持，那么返回的 HRESULT 是 S_OK。在一个成功的查询中，会发生两件事：一个指针被返回给由第二个参数（在这个例子中的 pMarshal）指定的地点，同时组件的引用计数加 1。引用计数加 1 归因于建立了组件的一个新连接。关闭连接要求调用 Release 一次：

```
pMarshal->Release ( ) ;
```

如果，在另一方面，所请求的接口不被支持，那么 QueryInterface 返回 E_NOINTERFACE。在这种情况下，不返回一个指针，引用计数也不会加 1。结果是 QueryInterface 函数用一个 NULL 的值覆盖返回值指针——在这里为 pMarshal——以保证它是一个已知的值。

这个查询功能的主要优点是一个 OLE 组件的客户可以对一个组件的功能进行一种运行时的查询。一个接口的存在意味着相关功能的存在，而一个接口不存在意味着该功能不存在。

和这一优点有关的是一个 OLE 组件的一种相对简单的升级。不过，新的功能从来不会通过扩展接口中的函数来加入；一旦定义了一个接口，那么这个接口所包含的特性将永远不会改变！新的特性是通过增加新的接口来加入的。现存的接口保持不受影响，继续对使用它们的客户提供支持。

20.1.5 OLE 是一种客户/服务器规范吗

刚开始使用 OLE 的程序员会问这么一个问题：OLE 是不是一种客户/服务器规范？你可以回顾一下第 17 章，DDE 是一种客户/服务器规范。事实上，OLE 版本 1.0 使用 DDE 作为它的传输机制。在那个版本中，用 DDE 消息来传送从 OLE 客户到 OLE 服务器的请求，而 OLE 服务器以更多地引用了 DDE 定义的全局（GlobalAlloc）存储的 DDE 消息来回答。

虽然“客户”和“服务器”经常在 OLE 1.0 的意义下使用，OLE 的发展使得这些术语只在一种通用的意义下使用。当两个组件进行交互时，二者通常同时既是客户（一个接口的使用者）又是服务器（一个接口的提供者）。尤其，在复合文档的含义下，术语“客户”被“容器应用程序”替代。在 OLE 动画环境中，使用术语“动画控件”来代替“动画客户”。虽然“服务器”仍然在使用，但使用了一个修饰性的前缀——“对象服务器”或者“动画服务器”——来指定所提供的服务所处的环境。

术语客户和服务器在试图理解一个特定的 OLE 接口时才是有用的。尤其是这些术语提供了一种有用的框架，来回答在编写 OLE 有关程序时经常会出现的一个问题：哪些接口是必须实现的？而哪些是由别人实现的？

20.2 私有组件服务器

现在是来看一看 IMALLOC 的时候了，这是一个动态链接库（DLL），演示了如何实

现一个接口。为了将重点放在接口的必要元素上，这个例子在几个方面加以了简化。首先，选择 OLE 的存储管理接口 IMalloc，因为大多数的程序员都已经理解了存储分配服务。其次，IMALLOC 在真正的存储工作方面做得很少，因为它将存储分配任务交给了 Win32 的私有堆函数（HeapCreate、HeapAlloc 等等）。最后，我们从一个私有组件——一个只能通过一种专有机制来访问的组件——开始，因为，你将在后面的 PUBMEM 中看到，使用一个公共组件将增加一些现在可以回避的复杂性。

虽然 IMALLOC 是一个简单的组件例子，但并不表示它不重要。尤其是它显示了定义任何一个 OLE 接口时所需要的元素，包括三个 IUnknown 接口函数：QueryInterface、AddRef 和 Release 的典型实现。它演示了使用特殊宏来创建可移植的接口定义，从而支持各种操作系统平台上（包括 Microsoft Windows 95、Microsoft Windows NT、Apple Macintosh 和可以容纳 OLE 的其它系统）的 C 和 C++。

私有组件是只能通过私有方式才可见的 OLE 组件。例如，IMALLOC 通过输出一个私有的入口点来提供对其接口的访问。这个例子是一个 DLL，但是私有组件的另一种用法应该是有许多私有 COM 组件组成的整个应用。主要的优点是组件客户和组件服务器之间的一种清晰界限所提供的封装。毕竟，定义一个 OLE 组件的是遵循 OLE 标准的接口创建——而不是接口的公共可用性。

在仔细阅读 IMALLOC 的源文件之后，你可能会认为它根本不是一个 OLE 组件。一个原因是，它没有调用任何 OLE 库。由于没有对 OleInitialize（或者 CoInitialize）的调用，你可能会奇怪 IMALLOC 怎么会被认为是一个 OLE 组件呢？原因很简单。IMALLOC 的接口是根据组件对象模型规范——一种 OLE 标准——建立的。由于它不要求 OLE 库服务，因此 IMALLOC 的实现不需要初始化 OLE 库，因而也避免了装入 OLE 库的负担。但是 IMALLOC 提供一种标准的 OLE 接口，当我们在本章后面的公共组件例子 PUBMEM 中使用这种接口时，这种标准接口将更加清楚。IMALLOC 的接口代码几乎不需要修改就可以在一个公共 OLE 组件中正确执行。

IMALLOC 是以 C++实现的，这对接口的实现是有意义的，因为一个 OLE 接口的二进制映象完全与 C++对象的映象完全匹配。虽然也有可能使用普通 C 来实现一个接口，但是它要求一些由 C++编译器自动完成的冗余步骤。为了了解在 C 中的 OLE 编程是什么样子，可以看一看 CALLER，它在本章后面出现。它是装入并调用 IMALLOC.DLL 的客户程序。IMALLOC 的程序文件如图 20-2 中所示。

IMALLOC.MAK

```
#----------------------
# IMALLOC.MAK make file
#----------------------

imalloc.dll : imalloc.obj
     $(LINKER) $(DLLFLAGS) -OUT:imalloc.dll imalloc.obj $(GUILIBS) uuid.lib

imalloc.obj : imalloc.cpp
     $(CC) $(CFLAGS) imalloc.cpp
```

IMALLOC.CPP

```
/*---------------------------------------
   IMALLOC.CPP -- Define an imalloc interface
                  (c) Paul Yao, 1996
  ---------------------------------------*/
#include <windows.h>
#include "imalloc.h"

//-------------------------------------------------------------
// CreateAllocator -- Exported function to create allocator
//-------------------------------------------------------------
EXPORT LPMALLOC CreateAllocator ()
     {
     DAlloc *pAllocator = new DAlloc () ;
     if (pAllocator != NULL && pAllocator->Initialize ())
          {
          pAllocator->AddRef () ;
          }
     else
          {
          delete pAllocator ;
          }

     return (LPMALLOC) pAllocator ;
     }

//-------------------------------------------------------------------
DAlloc::DAlloc ()
     {
     RefCount = 0 ;
     hHeap = NULL ;
     }

//-------------------------------------------------------------------
DAlloc::~DAlloc ()
     {
     if (hHeap)
          HeapDestroy (hHeap) ;
     }

//-------------------------------------------------------------------
BOOL DAlloc::Initialize ()
     {
     hHeap = HeapCreate (0, 4096, 65535) ;

     return (BOOL) hHeap ;
     }

//-------------------------------------------------------------------
STDMETHODIMP
DAlloc::QueryInterface (REFIID riid, LPVOID FAR *ppvObject)
     {
     // Always initialize "out" parameters to NULL
     *ppvObject = NULL ;

     // Everyone supports IUnknown
```

```
    if (riid == IID_IUnknown)
         *ppvObject = (LPUNKNOWN) this ;

    // We support IMalloc
    if (riid == IID_IMalloc)
         *ppvObject = (LPMALLOC) this ;

    if (*ppvObject == NULL)
         {
         // Interface not supported
         return E_NOINTERFACE ;
         }
    else
         {
         // Interface supported, so increment reference count
         ((LPUNKNOWN) *ppvObject)->AddRef () ;
         return S_OK ;
         }
    }

//-----------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::AddRef (void)
    {
    return ++RefCount ;
    }

//-----------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::Release (void)
    {
    if (0L != --RefCount)
         return RefCount ;

     delete this ;
     return 0L ;
    }

//-----------------------------------------------------------
STDMETHODIMP_ (void *)
DAlloc::Alloc (ULONG cb)
    {
    return HeapAlloc (hHeap, HEAP_ZERO_MEMORY, cb) ;
    }

//-----------------------------------------------------------
STDMETHODIMP_ (void *)
DAlloc::Realloc (void *pv, ULONG cb)
    {
    return HeapReAlloc (hHeap, HEAP_ZERO_MEMORY, pv, cb) ;
    }

//-----------------------------------------------------------
STDMETHODIMP_ (void)
DAlloc::Free (void *pv)
```

```
     {
     HeapFree (hHeap, 0, pv) ;
     }

//-------------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::GetSize (void *pv)
     {
     return HeapSize (hHeap, 0, pv) ;
     }

//-------------------------------------------------------------
STDMETHODIMP_ (int)
DAlloc::DidAlloc (void *pv)
     {
     PROCESS_HEAP_ENTRY phe ;

     ZeroMemory (&phe, sizeof (PROCESS_HEAP_ENTRY)) ;

     while (HeapWalk (hHeap, &phe))
          {
          if (phe.lpData == pv)
               return 1 ;
          }

     return 0 ;
     }

//-------------------------------------------------------------
STDMETHODIMP_ (void)
DAlloc::HeapMinimize (void)
     {
     HeapCompact (hHeap, 0) ;
     }
```

IMALLOC.H

```
//-------------------------------------------------------------
// C Interface to private allocator
//-------------------------------------------------------------
#define EXPORT extern "C" __declspec (dllexport)

EXPORT LPMALLOC CreateAllocator () ;

//-------------------------------------------------------------
// Implementation of allocator interface
//-------------------------------------------------------------
#undef  INTERFACE
#define INTERFACE DAlloc

DECLARE_INTERFACE_ (DAlloc, IMalloc)
     {
     // *** IUnknown methods **|
     STDMETHOD  (QueryInterface) (THIS_ REFIID riid, LPVOID FAR *ppv) ;
     STDMETHOD_ (ULONG, AddRef)  (THIS) ;
```

```
        STDMETHOD_ (ULONG, Release) (THIS) ;

        // *** IMalloc methods **|
        STDMETHOD_ (void *, Alloc)        (THIS_ ULONG cb) ;
        STDMETHOD_ (void *, Realloc)      (THIS_ void *pv, ULONG cb) ;
        STDMETHOD_ (void,   Free)         (THIS_ void *pv) ;
        STDMETHOD_ (ULONG,  GetSize)      (THIS_ void *pv) ;
        STDMETHOD_ (int,    DidAlloc)     (THIS_ void *pv) ;
        STDMETHOD_ (void,   HeapMinimize) (THIS) ;

#ifndef CINTERFACE
public :
        DAlloc () ;
        ~DAlloc () ;
        BOOL Initialize () ;

private :
        ULONG  RefCount ;
        HANDLE hHeap ;
#endif
        } ;
```

图 20-2 IMALLOC库

20.2.1 IMALLOC.DLL

作为私有的OLE组件，IMALLOC通过从私有输出函数CreateAllocator返回的接口指针来给出它的IMalloc接口。接口函数本身不需要被输出，因为返回的接口指针提供了对所有接口函数的完全访问。CreateAllocator分配DAlloc的一个实例，DAlloc是一个C++对象，对私有的Win32堆进行包装，它的虚函数表符合为IMalloc接口定义的二进制标准。DAlloc对象的指针是IMalloc接口指针，因此在CreateAllocator中的下面这种强制转换是合法的：

```
return (LPMALLOC) pAllocator ;
```

由于是在一个C++源文件中定义的，CreateAllocator要求特殊的处理来禁止C++名字混乱，并提供C调用的联编，这是由IMALLOC.H中的下列行完成的：

```
#define EXPORT extern "C" __declspec (dllexport)
EXPORT LPMALLOC CreateAllocator ( ) ;
```

EXPORT关键字和我们在第19章中定义的一样。

给C程序员的提示

IMALLOC是用C++写的，因此一些术语和约定可能是你不熟悉的。在CreateAllocator函数中，new操作符为一个DAlloc对象分配空间（通过调用

malloc），然后调用它的构造函数，DAlloc::DAlloc，以便执行对象数据成员的初始化。如果初始化失败则将从 CreateAllocator 中调用 delete 操作符，它在出现错误的情况下，释放对象分配的存储。它类似于在调用 free 后接着调用对象的析构函数，DAlloc::~DAlloc ( )。

作用域操作符（::)将函数指定为 C++类的成员。在 IMALLOC 中，这种表示法使 GetSize 函数成为 DAlloc 类的一个成员：

```
DAlloc::GetSize (...)
```

虽然所有带有前缀 DAlloc::的函数名都是 DAlloc 的成员，但是并非所有函数都在 DAlloc 的函数表中有入口点；只有那些使用 virtual 关键字定义的函数才这样。在 IMALLOC.H 中，STDMETHOD 和 STDMETHOD_宏（将在后面详细说明）包含 virtual 关键字。

有几句关于 IMalloc 与 DAlloc 之间区别的说明。作为一个服务合同，接口不能有一种实现直接和它相联系。以 C++的术语来说，这要求使所有的接口成为纯抽象类，也就是没有实现的纯接口。因此 DAlloc（“D”前缀代表“导出类”）使用 IMalloc 作为一个基类，从中继承了所有的虚函数成员。

但是由于三个额外的成员函数—— Dalloc 、~Dalloc 和 Initialize ——和两个额外的数据成员——RefCount 和 hHeap，DAlloc 是否还是 IMalloc 的一个合法实现呢？答案是“是”。三个额外的函数不是虚函数（即，它们没有使用 virtual 关键字定义），因此它们不会影响组成接口函数表的虚函数表。由于 DAlloc 的接口函数表和完成为 IMalloc 定义的服务所需要的表完全匹配，因此三个额外的函数是相同的。对于数据成员来说，这些都将被忽略，因为 OLE 接口被定义为无状态的（数据无关的）C++类。DAlloc 可以将它的数据成员用于它自己的目的，因为，一般说来，它们不会影响履行对 OLE 接口所要求的合同，或者具体一些，它们不会影响履行 IMalloc 接口所要求的合同。

20.2.2 那些古怪的宏

关于定义接口，刚开始可能令人迷惑的一个方面是一组宏的使用，如 DECLARE_INTERFACE 、STDMETHOD 和 THIS 等。使用这些宏的原因是使 OLE 代码——客户代码和服务器代码——可移植。这些宏以两种方式来增强 OLE 接口的可移植性：首先，这些宏既有一种 C 定义，又有一种 C++定义，这允许一个接口可以从两种语言中加以访问；其次，这些宏有助于将 OLE 移植到不同的操作系统平台。

在开发 OLE 时，第一个 C++编译器还刚刚可用。当时，C 和 C++比起来，C 是一种使用得更广泛的语言，因此 OLE 的创建者希望保证两种语言都可以使用。举一个例子将有助于理解这些宏是如何隐藏 C 和 C++之间区别的。使用宏，IUnknown 接口被定义如下：

```
#undef INTERFACE
#define INTERFACE IUnknown
```

```
DECLARE_INTERFACE (IUnknown)
    {
    STDMETHOD  (QueryInterface)  (THIS_
                                  REFIID riid,
                                  LPVOID FAR *ppvObj) PURE ;
    STDMETHOD_ (ULONG, AddRef)  (THIS) PURE ;
    STDMETHOD_ (ULONG, Release) (THIS) PURE ;
    } ;
```

许多接口定义宏有两个版本：普通版和增强版，增强版的名字有一个下划线(_)后缀。IUnknown 的定义给出了一对例子： STDMETHOD 和 STDMETHOD_。二者都将接口函数定义为 virtual 并使用一种特定的调用约定（ Win32 中的__stdcall ）。但是 STDMETHOD 指示了一个 HRESULT 的返回类型。其它返回类型必须被指定为 STDMETHOD_宏的一个参数。

下表总结了接口声明与定义的宏的作用：

宏名	描述
DECLARE_INTERFACE	定义一个没有基类的接口（例如，参见所附的 Iunknown 定义）。
DECLARE_INTERFACE_	定义一个具有基类的接口（例如，参见在 IMALLOC.CPP 中的 Dalloc ）。
PURE	在 C++中，对于纯（没有实现）虚函数的虚函数声明，在结束点增加一个“=0 ”。在 C 中，没有等价表示。
STDMETHOD	在接口声明中定义调用约定（ Win32 中的__stdcall ）和 HRESULT 返回类型。当为 C++进行编译时，这类函数也是虚函数，这意味着在 C++类虚函数表/OLE 接口函数表中将为它们分配一个位置。
STDMETHOD_	和上面的 STDMETHOD 一样，只是提供的返回类型不是 HRESULT 。
STDMETHODIMP	定义一个返回 HRESULT 的成员函数。
STDMETHODIMP_	定义一个返回非 HRESULT 类型的成员函数。
THIS	不要参数的接口函数的站位符。在 C++中， THIS 由 void 替换。在 C 中，THIS 由指向一个被引用的对象的指针替换，提供从 C 中调用一个 C++类的成员函数的正确联编。
THIS_	需要一个或者多个参数的接口函数的站位符。在 C++中， THIS_为空。在 C 中，它由上一个 THIS 项中讨论的指针替代，后面跟一个逗号。

当前面的 IUnknown 接口定义作为一种 C++定义让预处理器进行处理时，得到下列结果：

```
struct IUnknown
    {
    virtual HRESULT __stdcall QueryInterface (const IID *riid,
                                              void **ppv) = 0 ;
    virtual ULONG __stdcall AddRef (void) = 0 ;
    virtual ULONG __stdcall Release (void) = 0 ;
    }
```

你可能以为在整个定义的开始是 class 而不是 struct 。 struct 也是一个类，只是其成员

缺省为公共的。__stdcall 关键字定义栈结构的建立和清除。virtual 关键字定义一个虚函数，对于将在 C++虚函数表/OLE 接口函数表中包含的函数来说，这个关键字是必要的。最后，“=0 ”替代了 PURE 关键字，从而使整个类成为一个纯虚类——即，仅仅是接口的类，不能对它本身进行实例化。

为 IUnknown 产生的 C 定义更为复杂，因为其中创建了两个结构。尽管存在来自 C++代码的这种区别，当二者被编译成二进制代码时输出却是相同的。这是因为 OLE 的组件对象模型为接口连接定义了一种二进制标准。下面是相应的 IUnknown 的 C 代码：

```
typedef struct IUnknown
     {
     const struct IUnknownVtbl *lpVtbl ;
     } IUnknown ;
typedef const struct IUnknownVtbl IUnknownVtbl ;
const struct IUnknownVtbl
     {
     HRESULT (__stdcall * QueryInterface) (IUnknown *This,
                                           const IID *riid,
                                           void **ppd
     ULONG (__stdcall *,AddRef) (IUnknown *This) ;
     ULONG (__stdcall * Release) (IUnknown *This) ;
     } ;
```

其中的 C 实现为每个函数的参数列表的开始增加了一个 IUnknown *This 参数。这和总是被传递给 C++类成员函数的隐含的 this 指针相匹配。虽然似乎很奇怪，但是事实上有必要使 C 代码和 C++的实现相匹配。CALLER.EXE 的源文件——我们的下一个调用 IMALLOC.DLL 的例子和程序——显示了当一个 C 调用者引用一个 C++类函数时必须如何指定整个额外的参数。

平台之间的可移植性意味着对微妙的、平台有关的关键字加以防备。例如，Win16 OLE 接口要求用_far 和_pascal 关键字来创建正确的连接。在 Win32 中，这两个关键字被_stdcall 替代，作为所有接口函数的调用约定。当你在创建最终可能在一个 Apple Macintosh 系统、或者在 OLE 的 UNIX 实现上、或者在 OS/400 上、VMS 或者 MSV 上运行的 OLE 组件时，你可以使用宏为你无法预知的关键字提供站位符。一组平台有关的包含文件将匹配它们所需要的与环境有关的关键字。

20.2.3 由 IUnknown 提供的服务

虽然 IMALLOC.DLL 演示了 IMalloc 接口的一种实现，但它最令人感兴趣的特色是它对三个 IUnknown 成员函数的实现和这些函数提供的两种服务：对组件生命周期的管理和对多接口组件的支持。

20.2.3.1 管理组件生命周期

一个 OLE 组件的生命是由一个私有数据成员和两个接口函数决定的。数据成员是引用计数——在 DAlloc 中名为 RefCount 的一个 ULONG 值——而两个接口函数是 AddRef 和 Release 。

当调用私有的组件创建函数 CreateAllocator 时， IMALLOC 创建一个分配器组件。在 CreateAllocator 内的相关代码行如下所示：

```
Dalloc * pAllocator = new Dalloc ( ) ;
if (pAllocator != NULL && pAllocator->Initialize ( ))
    {
    pAllocator->AddRef ( ) ;
    }
else
   {
   delete pAllocator ;
   }
```

当对象被创建时，它的引用计数在 DAlloc 的构造函数中被初始化为 0 。同时， Win32 堆的句柄被初始化为 NULL：

```
DAlloc::DAlloc ( )
    {
    RefCount = 0 ;
    hHeap = NULL ;
    }
```

在 C++中，通过将类构造函数中的工作限制为初始化数据成员从而简化了代码，正如上面所看到的。原因是构造函数没有方便的方法来提示对象创建者失败的情况，因此在其中只执行不会失败的工作。可能会失败的任务通常被分派给特定的初始化函数，如 DAlloc 的 Initialize 函数，它创建 Win32 堆，来满足 IMalloc 的分配请求。

一旦 CreateAllocator 成功地创建并初始化一个 DAlloc 对象之后，它通过调用 AddRef 将引用计数加 1 。整个函数定义如下：

```
STDMETHODIMP_ (ULONG)
DAlloc::AddRef (void)
    {
    return ++RefCount ;
    }
```

现在你看到了为什么整个函数返回一个非 HRESULT 的值：让调用者可以知道引用计数的当前值（但这只能用于调试目的）。

在 IMALLOC 中对 AddRef 的其它调用发生在 QueryInterface 。当 QueryInterface 返回指向一个组件接口的指针时，它调用 AddRef ，从而正确地维持从组件到外部世界的连接计数。

虽然在 IMALLOC 中有两个地方都调用了 AddRef，但是却没有对 Release 的调用。只有组件客户才调用这个函数，如果该客户可以结束一个组件的生命，这样做是有意义的。下面是整个函数的核心代码：

```
STDMETHODIMP_ (ULONG)
DAlloc::Release ( )
    {
    if (0L != --RefCount)
        return RefCount ;
```

```
    delete this ;
    return 0L ;
    }
```

当引用计数变为 0 时，调用 delete 操作符使组件释放自己并调用它的析构函数，DAlloc::~DAlloc 。在这个函数中， Win32 的堆被清除：

```
if (hHeap)
    HeapDestroy (hHeap) ;
```

这清楚地表明：将引用计数减少到 0——即，释放对象——会有不可逆转的影响。

虽然一个 OLE 组件的 IUnknown 成员提供了管理组件生命周期的方法，但是最终还有赖于组件的客户在正确的时间正确地调用这些成员函数。对 Release 过少的调用将导致组件占用不必要的存储。而过多的调用导致对象在不应该消失的时候消失。显然，保证引用计数完全正确是客户软件需要做的一项重要工作。

通过这里对 IUnknown 成员函数实现的研究，应该得出一些通用的规则。首先，对 AddRef 的每个调用都必须有一个 Release 调用和它匹配。由于 QueryInterface 调用 AddRef，因此 QueryInterface 也必须有一个 Release 调用和它匹配。

20.2.3.2 支持多接口组件

从 OLE 接口服务器的角度来看， QueryInterface 函数是组件负责分发连接请求的中心转接板。下面是从 IMALLOC.CPP 中摘出的 QueryInterface 函数：

```
STDMETHODIMP
DAlloc::QueryInterface (REFIID riid, LPVOID FAR *ppvObject)
     {
     // Always initialize "out" parameters to NULL
     *ppvObject = NULL ;

     // Everyone supports IUnknown
     if (riid == IID_IUnknown)
          *ppvObject = (LPUNKNOWN) this ;

     // We support IMalloc
     if (riid == IID_IMalloc)
          *ppvObject = (LPMALLOC) this ;

     if (*ppvObject == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE ;
          }
     else
          {
          // Interface supported, so increment reference count
          ((LPUNKNOWN) *ppvObject)->AddRef () ;
          return S_OK ;
          }
     }
```

和大多数 OLE 接口函数一样， QueryInterface 返回 HRESULT 。两个返回值被定义来指示没有意外的错误发生： S_OK 意味着组件支持所请求的接口，并且返回指向该接口的一个指针； E_NOINTERFACE 意味着所请求的接口不被支持，并且返回一个 NULL 指针。

由于对跨进程请求的支持，因此， OLE 函数参数和接口参数被分为 in 、 out 和 in/out 三类。相对于被调用的函数，“ in ” 参数只有一个方向的输入数据，“ out ” 参数只有一个方向的输出数据，而 “ in/out ” 参数既提供输入数据又提供输出数据。这种简单的分类有助于减少需要跨进程或网络界限传送的数据量。单向参数——“ in ” 和 “ out ” ——只需在一半的时间传送，而双向参数 in/out 却必须既在函数调用前又在函数调用后被拷贝。

QueryInterface 有一个 “ in ” 参数 (riid)和一个 “ out ” 参数 (ppvObject)。为了保持一致性， COM 规范规定：任何作为指针的 out 参数必须被设置为已知值。这就是为什么在 QueryInterface 的第一行将 ppvObject 设置为 NULL 的原因：

```
// 总是将 “ out ” 参数初始化为 NULL
*ppvObject = NULL ;
```

如果所请求的接口不被支持，那么 NULL 是正确的指针值。另一方面，如果一个接口被支持，那么 NULL 将被指向所请求接口的一个指针覆盖掉。

当接口被支持时，指向该接口的指针被返回。然后，为了保证该接口引用计数的正确性，应该调用该接口的 AddRef 成员：

```
((LPUNKNOWN) *ppvObject)->AddRef ( ) ;
```

认识到这一点对于 OLE 组件的客户尤其重要，因为对 QueryInterface 的成功调用总是需要有 Release 调用与之匹配，从而让组件知道何时可以安全地终止自己。

虽然 IMALLOC.DLL 只支持一种接口，但事实上 QueryInterface 响应却似乎支持两个接口： IUnknown 和 IMalloc 。从客户的角度来看，确实有两个接口被支持：一个控制底层的组件，另一个提供存储分配服务。二者使用同一个虚函数表这一事实是 COM 体系结构的全局性设计抉择。

想象一下 QueryInterface 是如何提供额外的接口连接的，这并不太难。 QueryInterface 所采用机制的原理是直观的。增加一个 if 语句来检查特定的接口 ID ，从而可以支持任意数目的新接口，比如下面的例子中的 IID_IMarshal :

```
// 我们支持 Imarshal
if (riid == IID_IMarshal)
    *ppvObject = (LPMARSHAL) pMarshal ;
```

不那么直观明显的是由 pMarshal 指向的接口和一个特定的 DAlloc::QueryInterface 函数之间的关系。对于这个问题，有几种回答，但是都不在这里的讨论范围之内。简单地说，支持 IMarshal 接口——可以称之为 DMarshal ——的 C++对象可以被包含在 DAlloc 中，或者，通过多重继承，多个被支持的接口可以共享一组通用的 IUnknown 函数。无论选择哪种特定的实现，对于组件客户来看是一样的： QueryInterface 提供中心转接控制，建立与组件所支持的接口之间的连接。

为了了解客户可以如何利用 IMALLOC 的私有组件，我们来看看 CALLER 。它还表明

如何用直接的 C 代码来访问 OLE 服务。

20.3 私有组件的客户

CALLER.EXE 程序显示了私有组件的客户如何建立接口连接，如何调用接口函数，以及如何切断和接口的连接。 IMALLOC 从服务器的角度显示了接口创建的细节，而 CALLER 从客户的角度来显示接口使用的细节。在二者中，客户的角度更容易掌握。除了和正确管理服务器生命周期——由 IUnknown 接口函数提供——有关的问题之外，对使用过函数库的任何程序员而言，客户这一侧的事情都应该是非常熟悉的：指定所请求的函数，选择适当的参数，并检查返回值是成功还是失败。

COM 接口和其它编程接口（诸如 Windows 的 Win32 API 或者 C 运行库）在一个方面是不同的：其它的 API 提供一组函数，这些函数可能是通过名字（如对 strcpy 、 strlen 、 strcat 等等），或者是通过第一个参数（如对 MoveWindow 、 FindWindow 、 ShowWindow 等等）相互关联的，而 COM 接口函数定义的是和其它函数集之间的清晰界限。在对象服务器中，这一点反映在所有接口函数都是同一接口的虚函数这一事实上。在一个接口的客户中，所有接口函数都是由指向接口函数的一个通用指针来访问的这一事实，表明了这一关联。

CALLER.EXE 还显示了如何从 C 中访问 COM 接口。这需要稍微多花一些工夫——多敲一点程序。例如，下面是访问一个 IMalloc 接口的 Alloc 成员的 C++代码：

```
LPSTR pData = (LPSTR)pMalloc->Alloc (cbdata) ;
```

而 C 的等价调用如下所示：

```
LPSTR pData = (LPSTR)pMalloc->lpVtbl->Alloc (pMalloc, cbdata) ;
```

虽然不用多敲太多键，但是确实需要多做些事情，因而也更容易发生错误。

一种显然的解决方案是将这样的调用包装为一组接口特定的宏。对于准备以 OLE 接口来向后兼容庞大的 C 应用程序的开发小组来说，这是有帮助的。例如，就前面对 IMalloc::Alloc 的调用，可以定义一个如下所示的包装宏：

```
#ifdef __cplusplus
#define ALLOC (pInt, cbSize) (pInt##->Alloc (cbSize##)
#else
#define ALLOC (pInt, cbSize) \
            (pInt##->lpVtbl->Alloc (pInt##, cbSize##))
#endif
```

这样，无论从 C 或者 C++中，调用 IMalloc::Alloc 的语法都是一样的：

```
LPSTR pData = (LPSTR)ALLOC (pMalloc, cbdata) ;
```

这样的调用没有为 C++程序员省去多少功夫（而且引起更多的困惑，因为这是访问一个

C++类的非标准方式）。但是，对于 C 程序员来说，这样的一组宏使他们避免了多余的键入动作。

CALLER 通过分配存储并将值存储到 10 个容器之一演示了对 IMALLOC 接口的调用。如图 20-3 所示，CALLER 显示了其主窗口中每个槽的内容。为了增加趣味性，CALLER 使用两个分配器：一个通过 IMALLOC.DLL 提供的 IMalloc 接口来访问，而另一个是常规的 C 运行库函数。图 20-4 显示了 CALLER 源文件。

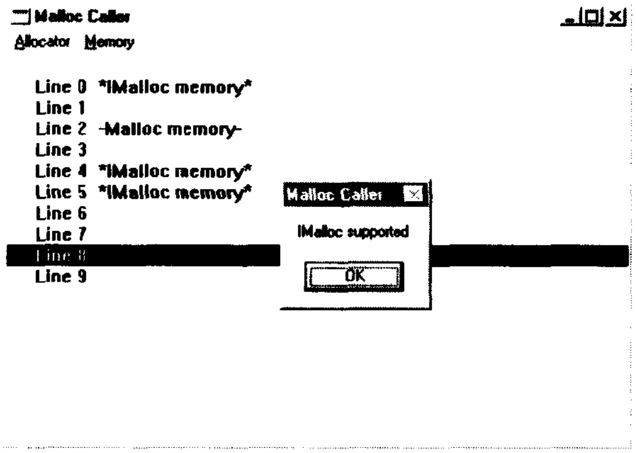


图 20-3 CALLER的显示

CALLER.MAK

```
#---------------------
# CALLER.MAK make file
#---------------------

caller.exe : caller.obj caller.res imalloc.lib
     $(LINKER) $(GUIFLAGS) -OUT:caller.exe caller.obj caller.res \
               imalloc.lib $(GUILIBS) uuid.lib

caller.obj : caller.c caller.h imalloc.h
     $(CC) -DCINTERFACE $(CFLAGS) caller.c

caller.res : caller.rc caller.h
     $(RC) $(RCVARS) caller.rc
```

CALLER.C

```
/*--------------------------------------------
   CALLER.C -- Call into private OLE component
               (c) Paul Yao, 1996
  --------------------------------------------*/
#include <windows.h>
#include "caller.h"
#include "imalloc.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
```

```
char szWndClass[] = "CallerWindow" ;
char szAppName[]  = "Malloc Caller" ;

//-------------------------------------------------------------------
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wc ;

     wc.cbSize         = sizeof (wc) ;
     wc.lpszClassName  = szWndClass ;
     wc.hInstance      = hInstance ;
     wc.lpfnWndProc    = WndProc ;
     wc.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
     wc.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
     wc.lpszMenuName   = "MAIN" ;
     wc.hbrBackground  = (HBRUSH) (COLOR_WINDOW + 1) ;
     wc.style          = 0 ;
     wc.cbClsExtra     = 0 ;
     wc.cbWndExtra     = 0 ;
     wc.hIconSm        = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wc) ;

     hwnd = CreateWindowEx (0L, szWndClass, szAppName,
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, cmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }
     return msg.wParam ;
     }

//-------------------------------------------------------------------
LRESULT CALLBACK
WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int        iCurLine = 0 ;
     static LPMALLOC   pMalloc = NULL ;
     static LPSTR      szLine[10] ;
     static RECT       rHit[10] ;

     switch (iMsg)
          {
          case WM_CREATE :
```

```
          // Initialize data pointer array
          ZeroMemory (szLine, sizeof (szLine)) ;
          return 0 ;

     case WM_COMMAND :
          switch (LOWORD (wParam))
               {
               case IDM_CREATE :
                    pMalloc = CreateAllocator () ;
                    if (pMalloc == NULL)
                         {
                         MessageBox (hwnd, "Error: No allocator",
                                     szAppName, MB_OK) ;
                         return 0 ;
                         }

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;

               case IDM_DESTROY :
                    {
                    int i ;

                    // Mark allocated blocks as invalid
                    for (i = 0 ; i < 10 ; i++)
                         {
                         if ((szLine[i] != NULL) &&
                             (pMalloc->lpVtbl->DidAlloc (pMalloc,
                                                          szLine[i])))
                              {
                              szLine[i] = NULL ;
                              }
                         }

                    // Disconnect from & free allocator
                    pMalloc->lpVtbl->Release (pMalloc) ;
                    pMalloc = NULL ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;
                    }

               case IDM_IUNKNOWN :
                    {
                    LPUNKNOWN pUnk ;
                    HRESULT hr =
                         pMalloc->lpVtbl->QueryInterface (pMalloc,
                                                          IID_IUnknown,
                                                          (void **)&pUnk) ;
                    if (SUCCEEDED (hr))
                         {
                         pUnk->lpVtbl->Release (pUnk) ;
                         MessageBox (hwnd, "IUnknown supported",
                                     szAppName, MB_OK) ;
                         }
```

```
          else
               {
               MessageBox (hwnd, "IUnknown not supported",
                           szAppName, MB_OK) ;
               }
          return 0 ;
          }

     case IDM_IMALLOC :
          {
          LPUNKNOWN pUnk ;
          HRESULT hr =
               pMalloc->lpVtbl->QueryInterface (pMalloc,
                                              IID_IMalloc,
                                              (void **)&pUnk) ;
          if (SUCCEEDED (hr))
               {
               pUnk->lpVtbl->Release (pUnk) ;
               MessageBox (hwnd, "IMalloc supported",
                           szAppName, MB_OK) ;
               }
          else
               {
               MessageBox (hwnd, "IMalloc not supported",
                           szAppName, MB_OK) ;
               }
          return 0 ;
          }

     case IDM_IMARSHAL :
          {
          LPUNKNOWN pUnk ;
          HRESULT hr =
               pMalloc->lpVtbl->QueryInterface (pMalloc,
                                              IID_IMarshal,
                                              (void **)&pUnk) ;
          if (SUCCEEDED (hr))
               {
               pUnk->lpVtbl->Release (pUnk) ;
               MessageBox (hwnd, "IMarshal supported",
                           szAppName, MB_OK) ;
               }
          else
               {
               MessageBox (hwnd, "IMarshal not supported",
                           szAppName, MB_OK) ;
               }
          return 0 ;
          }

     case IDM_ALLOCATE_CUSTOM :
          if (szLine[iCurLine] != NULL)
               {
               MessageBox (hwnd, "Error: Free First",
                           szAppName, MB_OK) ;
```

```
                         return 0 ;
                         }

                    // Allocate from IAllocate interface
                    szLine[iCurLine] =
                         (char *) pMalloc->lpVtbl->Alloc (pMalloc, 100) ;
                    lstrcpy (szLine[iCurLine], "*IMalloc memory*") ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;

               case IDM_ALLOCATE_DEFAULT :
                    if (szLine[iCurLine] != NULL)
                         {
                         MessageBox (hwnd, "Error: Free First",
                                        szAppName, MB_OK) ;
                         return 0 ;
                         }

                    // Allocate from default heap
                    szLine[iCurLine] = (char *) malloc (100) ;
                    lstrcpy (szLine[iCurLine], "-Malloc memory-") ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;

               case IDM_FREE :
                    if (szLine[iCurLine] == NULL)
                         {
                         MessageBox (hwnd, "Error: Nothing to free",
                                        szAppName, MB_OK) ;
                         return 0 ;
                         }

                    if (pMalloc == NULL)
                         {
                         goto FreeMalloc ;
                         }

                    // Free allocated object
                    if (pMalloc->lpVtbl->DidAlloc (pMalloc,
                                                   szLine[iCurLine]))
                         {
                         pMalloc->lpVtbl->Free (pMalloc,
                                                szLine[iCurLine]) ;
                         }
                    else
                         {
FreeMalloc:
                         free (szLine[iCurLine]) ;
                         }

                    szLine[iCurLine] = NULL ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
```

```
                    return 0 ;
                    }

          case WM_DESTROY :
               // Disconnect from & free allocator
               if (pMalloc)
                    {
                    pMalloc->lpVtbl->Release (pMalloc) ;
                    pMalloc = NULL ;
                    }

               PostQuitMessage (0) ;  // Handle application shutdown
               return 0 ;

          case WM_INITMENU :
               {
               HMENU hMenu = (HMENU) wParam ;
               if (pMalloc)
                    {
                    EnableMenuItem (hMenu, IDM_CREATE,          MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_DESTROY,         MF_ENABLED) ;
                    EnableMenuItem (hMenu, IDM_ALLOCATE_CUSTOM, MF_ENABLED) ;
                    EnableMenuItem (hMenu, IDM_IUNKNOWN,        MF_ENABLED) ;
                    EnableMenuItem (hMenu, IDM_IMALLOC,         MF_ENABLED) ;
                    EnableMenuItem (hMenu, IDM_IMARSHAL,        MF_ENABLED) ;
                    }
               else
                    {
                    EnableMenuItem (hMenu, IDM_CREATE,          MF_ENABLED) ;
                    EnableMenuItem (hMenu, IDM_DESTROY,         MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_ALLOCATE_CUSTOM, MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_IUNKNOWN,        MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_IMALLOC,         MF_GRAYED) ;
                    EnableMenuItem (hMenu, IDM_IMARSHAL,        MF_GRAYED) ;
                    }
               return 0 ;
               }

          case WM_LBUTTONDOWN :
               {
               int i ;
               int x = LOWORD (lParam) ;
               int y = HIWORD (lParam) ;
               POINT pt = { x, y } ;

               for (i = 0 ; i < 10 ; i++)
                    {
                    if (PtInRect (&rHit[i], pt))
                         {
                         if (iCurLine != i)  // Minimize screen blink
                              {
                              InvalidateRect (hwnd, &rHit[iCurLine], TRUE) ;
                              InvalidateRect (hwnd, &rHit[i], TRUE) ;
                              iCurLine = i ;
                              }
```

```
                    break ;
                    }
               }
               return 0 ;
          }

     case WM_PAINT :
          {
          char          szBuff[10] ;
          COLORREF      crText, crBack ;
          HDC           hdc ;
          int           cc ;
          int           i ;
          int           XCount, XText, Y ;
          int           cyLineHeight ;
          PAINTSTRUCT   ps ;
          RECT          rOpaque ;
          TEXTMETRIC    tm ;

          hdc = BeginPaint (hwnd, &ps) ;

          // Fetch line height
          GetTextMetrics (ps.hdc, &tm) ;
          cyLineHeight = tm.tmHeight + tm.tmExternalLeading ;

          // Fetch current text colors
          crText = GetTextColor (ps.hdc) ;
          crBack = GetBkColor (ps.hdc) ;

          XCount = tm.tmAveCharWidth * 3 ;
          XText  = XCount + tm.tmAveCharWidth * 7 ;
          Y      = tm.tmHeight ;

          for (i = 0 ; i < 10 ; i++, Y += cyLineHeight)
               {
               // Set colors to highlight current line
               if (i == iCurLine)
                    {
                    SetTextColor (ps.hdc, crBack) ;
                    SetBkColor (ps.hdc, crText) ;

                    SetRect (&rOpaque, 0, Y, 9999, Y + cyLineHeight) ;
                    ExtTextOut (ps.hdc, 0, 0, ETO_OPAQUE, &rOpaque,
                                NULL, 0, NULL ) ;
                    }
               else
                    {
                    SetTextColor (ps.hdc, crText) ;
                    SetBkColor (ps.hdc, crBack) ;
                    }

               // Display line count
               cc = wsprintf (szBuff, "Line %d", i) ;
               TextOut (ps.hdc, XCount, Y, szBuff, cc) ;

               // Display text if a string has been defined
               if (szLine[i] != NULL)
```

```
                    {
                    cc = lstrlen (szLine[i]) ;
                    TextOut (ps.hdc, XText, Y, szLine[i], cc) ;
                    }

               // Calculate hit test rectangle
               SetRect (&rHit[i], 0, Y, 9999, Y + cyLineHeight) ;
               }

          EndPaint (hwnd, &ps) ;
          return 0 ;
          }
     default :
          return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
}
```

CALLER.H

```
#define IDM_CREATE            1
#define IDM_DESTROY           2
#define IDM_IUNKNOWN          3
#define IDM_IMALLOC           4
#define IDM_IMARSHAL          5
#define IDM_ALLOCATE_CUSTOM   6
#define IDM_ALLOCATE_DEFAULT  7
#define IDM_FREE              8
#define IDM_CHECK             9
```

CALLER.RC

```
#include "callpub.h"

MAIN MENU
     {
     POPUP "&Allocator"
          {
          MENUITEM "&Create",                          IDM_CREATE
          MENUITEM "&Destroy",                         IDM_DESTROY
          MENUITEM SEPARATOR
          MENUITEM "QueryInterface IID_IUnknown",  IDM_IUNKNOWN
          MENUITEM "QueryInterface IID_IMalloc",   IDM_IMALLOC
          MENUITEM "QueryInterface IID_IMarshal",  IDM_IMARSHAL
          }

     POPUP "&Memory"
          {
          MENUITEM "&Allocate (IMalloc)",  IDM_ALLOCATE_CUSTOM
          MENUITEM "&Allocate (malloc)",   IDM_ALLOCATE_DEFAULT
          MENUITEM "&Free",                IDM_FREE
          }
     }
```

图 20-4 CALLER程序

CALLER 没有直接使用 OLE 库，因此，像它之前的 IMALLOC.DLL 一样， CALLER 没有初始化 OLE 库。（当 CALLER 在本章后面被升级为 CALLPUB 时，这一点就变了；

正如 IMALLOC 的后代 PUBMEM 一样， CALLPUB 也必须依赖于 OLE 库，因而必须使用适当的初始化和终止函数。）

CALLER 对 IMalloc 接口的所有访问都是通过菜单选择进行的，这使你完全控制调用什么以及何时调用。运行 CALLER 时，要做的第一件事是通过选择 Allocator 菜单中的 Create ，创建一个分配组件。这将调用 IMALLOC 的 CreateAllocator 函数，该函数返回访问 IMALLOC 的分配器接口所需要的接口指针。

Allocator 菜单中的 Destroy 命令使分配组件调用 Release 将它的引用计数减 1 ，并由此清除分配组件：

```
pMalloc->lpVtbl->Release (pMalloc) ;
pMalloc = NULL ;
```

将刚被释放的一个接口指针的值设置为 NULL 是一种好习惯，这保证了你再次使用它时会导致一个异常——在你发行软件之前有希望被发现的错误。

CALLER 使用分配器的 QueryInterface 成员来检查其所支持的接口。在客户这边，每当调用一次 QueryInterface 来提供一个接口指针时，总要调用 Release 来进行释放。这是很重要的，否则，你可能会将引用计数大于 0 的一个组件留在存储中，而该组件的客户却认为它已经被终止了。

下面，我们将注意力转到创建公共组件服务器所涉及的方方面面。

20.4　公共组件的服务器

PUBMEM.DLL 演示了公共 OLE 组件服务器的创建。公共和私有组件之间的差别不在于接口本身；事实上，由 PUBMEM 提供的公共 IMalloc 接口和前面的 IMALLOC 实例提供的接口是相同的（只有一个小小的改变）。公共 OLE 组件是从公共 OLE 组件类中创建的组件。

一个公共 OLE 组件类要求几个因素之间的相互协调。一个组件类有唯一的类标识符（ CLSID ）和一个将类 ID 和已安装的 EXE 和 DLL 联系在一起的登记表(registry)入口。组件本身必须提供叫做类工厂（ class factory ）的标准机制来根据请求创建一个组件。类工厂自身是一个私有 OLE 组件，由对 DLL 服务器的 DllGetClassObject 输出入口点的调用来创建。这一入口点为 IClassFactory 接口提供服务。 OLE 库周期性地调用另一个入口点 DllCanUnloadNow ，以便确认 DLL 是否可以安全地终止其所容纳的组件。虽然比私有组件复杂，但是类工厂为输出多种类型的组件类提供了通用的机制，而每一种组件类又都可以提供多个不同的接口。 PUBMEM 的源文件如图 20-5 所示。

PUBMEM.MAK

```
#----------------------
# PUBMEM.MAK make file
#----------------------
```

```
pubmem.dll : pubmem.obj classfac.obj compobj.obj
     $(LINKER) /EXPORT:DllGetClassObject /EXPORT:DllCanUnloadNow \
     $(DLLFLAGS) -OUT:pubmem.dll pubmem.obj \
     classfac.obj compobj.obj $(GUILIBS) uuid.lib ole32.lib

pubmem.obj : pubmem.cpp pubmem.h
     $(CC) $(CFLAGS) pubmem.cpp

classfac.obj : classfac.cpp pubmem.h
     $(CC) $(CFLAGS) classfac.cpp

compobj.obj : compobj.cpp pubmem.h
     $(CC) $(CFLAGS) compobj.cpp
```

PUBMEM.CPP

```
/*----------------------------------------------
   PUBMEM.CPP -- Define a public imalloc component
                 (c) Paul Yao, 1996
  ----------------------------------------------*/
#include <windows.h>
#include "pubmem.h"

extern int cObject ;

//-------------------------------------------------------------------
// CreateAllocator -- Exported function to create allocator
//-------------------------------------------------------------------
EXPORT LPMALLOC CreateAllocator ()
     {
     DAlloc *pAllocator = NULL ;

     pAllocator = new DAlloc () ;
     if (pAllocator != NULL && pAllocator->Initialize ())
          {
          pAllocator->AddRef () ;
          }
     else
          {
          delete pAllocator ;
          }

     return (LPMALLOC) pAllocator ;
     }

//-------------------------------------------------------------------
DAlloc::DAlloc ()
     {
     RefCount = 0 ;
     hHeap = NULL ;
     }

//-------------------------------------------------------------------
```

```
DAlloc::~DAlloc ()
     {
     if (hHeap)
          HeapDestroy (hHeap) ;
     }

//-----------------------------------------------------------------------
BOOL DAlloc::Initialize ()
     {
     hHeap = HeapCreate (0, 4096, 65535) ;
     return (BOOL) hHeap ;
     }

//-----------------------------------------------------------------------
STDMETHODIMP
DAlloc::QueryInterface (REFIID riid, LPVOID FAR *ppvObject)
     {
     // Always initialize "out" parameters to NULL
     *ppvObject = NULL ;

     // Everyone supports IUnknown
     if (riid == IID_IUnknown)
          *ppvObject = (LPUNKNOWN) this ;

     // We support IMalloc
     if (riid == IID_IMalloc)
          *ppvObject = (LPMALLOC) this ;

     if (*ppvObject == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE ;
          }
     else
          {
          // Interface supported, so increment reference count
          ((LPUNKNOWN) *ppvObject )->AddRef () ;
          return S_OK ;
          }
     }

//-----------------------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::AddRef (void)
     {
     return ++RefCount ;
     }

//-----------------------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::Release (void)
     {
     if (0L != RefCount)
          return RefCount ;
```

```
     --cObject ;
     delete this ;
     return 0L ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (void *)
DAlloc::Alloc (ULONG cb)
     {
     return HeapAlloc (hHeap, HEAP_ZERO_MEMORY, cb) ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (void *)
DAlloc::Realloc (void *pv, ULONG cb)
     {
     return HeapReAlloc (hHeap, HEAP_ZERO_MEMORY, pv, cb) ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (void)
DAlloc::Free (void *pv)
     {
     HeapFree (hHeap, 0, pv) ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (ULONG)
DAlloc::GetSize (void *pv)
     {
     return HeapSize (hHeap, 0, pv) ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (int)
DAlloc::DidAlloc (void *pv)
     {
     PROCESS_HEAP_ENTRY phe ;
     ZeroMemory (&phe, sizeof (PROCESS_HEAP_ENTRY)) ;

     while (HeapWalk (hHeap, &phe))
          {
          if (p he.lpData == pv)
               return 1 ;
          }

     return 0 ;
     }

//--------------------------------------------------------------------
STDMETHODIMP_ (void)
DAlloc::HeapMinimize (void)
     {
     HeapCompact (hHeap, 0) ;
     }
```

CLASSFAC.CPP

```
/*------------------------------------------
   CLASSFAC.CPP -- OLE Class Factory component
                   (c) Paul Yao, 1996
  ------------------------------------------*/
#include <windows.h>
#include <initguid.h>
#include "pubmem.h"

extern int cObject ;
extern int cLockCount ;

//-------------------------------------------------------------------
DClassFactory::DClassFactory ()
     {
     RefCount = 0 ;
     }

//-------------------------------------------------------------------
DClassFactory::~DClassFactory ()
     {
     }

//-------------------------------------------------------------------
STDMETHODIMP
DClassFactory::QueryInterface (REFIID riid, LPVOID FAR *ppvObj)
     {
     // Init recipient's pointer
     *ppvObj = NULL ;

     // If asking for IUnknown, we can provide
     if (riid == IID_IUnknown)
          *ppvObj = (LPUNKNOWN) this ;

     // If asking for IClassFactory, we can provide
     if (riid == IID_IClassFactory)
          *ppvObj = (LPCLASSFACTORY) this ;

     // Make sure reference count reflects access
     if (*ppvObj == NULL)
          {
          // Interface not supported
          return E_NOINTERFACE ;
          }
     else
          {
          // Interface supported to increment reference count
          ((LPUNKNOWN) *ppvObj)->AddRef () ;
          return S_OK ;
          }
     }

//-------------------------------------------------------------------
STDMETHODIMP_ (ULONG)
```

```
DClassFactory::AddRef ()
     {
     return ++RefCount ;
     }

//------------------------------------------------------------------
STDMETHODIMP_ (ULONG)
DClassFactory::Release ()
     {
     if (0L != --RefCount)
          return RefCount ;

     delete this ;
     return 0L ;
     }

//------------------------------------------------------------------
STDMETHODIMP
DClassFactory::CreateInstance (LPUNKNOWN pUnkOuter, REFIID riid,
                               LPVOID FAR *ppvObject)
     {
     // Initialize return pointer
     *ppvObject = NULL ;

     // If trying to aggregate, fail
     if (pUnkOuter != NULL)
          return CLASS_E_NOAGGREGATION ;

     // Create memory allocation object
     LPMALLOC pMalloc = CreateAllocator () ;

     if (pMalloc == NULL)
          {
          return E_OUTOFMEMORY ;
          }
     else
          {
          // Fetch interface requested by caller
          HRESULT hr = pMalloc->QueryInterface (riid, ppvObject) ;

          // Decrement reference count produced by CreateAllocator
          pMalloc->Release () ;

          // Increment count of objects
          if (SUCCEEDED (hr))
               ++cObject ;

          return hr ;
          }
     }

//------------------------------------------------------------------
STDMETHODIMP
DClassFactory::LockServer (BOOL fLock)
     {
```

```
    if (fLock)
        {
        ++cLockCount ;
        }
    else
        {
        --cLockCount ;
        }
    return NOERROR ;
    }
```

COMPOBJ.CPP

```
/*-----------------------------------------------
   COMPOBJ.CPP -- Component Object registration
                  (c) Paul Yao, 1996
  -----------------------------------------------*/
#include <windows.h>
#include "pubmem.h"

int cObject    = 0 ;
int cLockCount = 0 ;

//-------------------------------------------------------------------
HRESULT APIENTRY
DllGetClassObject (REFCLSID rclsid, REFIID riid, LPVOID *ppvObj)
     {
     // Initialize "out" pointer to known value
     *ppvObj = NULL ;

     if (rclsid != CLSID_ALLOCATOR)
          {
          return CLASS_E_CLASSNOTAVAILABLE ;
          }

     DClassFactory *pClassFactory = new DClassFactory () ;
     if (pClassFactory == NULL)
          {
          return E_OUTOFMEMORY ;
          }
     else
          {
          return pClassFactory->QueryInterface (riid, ppvObj) ;
          }
     }

//-------------------------------------------------------------------
HRESULT APIENTRY
DllCanUnloadNow (void)
     {
     if (cObject > 0 || cLockCount > 0)
          {
          return S_FALSE ;
          }
     else
```

```
          {
          return S_OK ;
          }
     }
```

PUBMEM.H

```
//------------------------------------------------------------------
// C Interface to private allocator
//------------------------------------------------------------------
#define EXPORT extern "C" __declspec (dllexport)

EXPORT LPMALLOC CreateAllocator () ;

// {308D0430-1090-11cf-B92A-00AA006238F8}
DEFINE_GUID (CLSID_ALLOCATOR,
     0x308d0430, 0x1090, 0x11cf, 0xb9,
     0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8) ;

//------------------------------------------------------------------
// Implementation of allocator interface
//------------------------------------------------------------------
#undef  INTERFACE
#define INTERFACE DAlloc

DECLARE_INTERFACE_ (DAlloc, IMalloc)
     {
     // *** IUnknown methods **|
     STDMETHOD  (QueryInterface) (THIS_ REFIID riid, LPVOID FAR *ppv) ;
     STDMETHOD_ (ULONG, AddRef)  (THIS) ;
     STDMETHOD_ (ULONG, Release) (THIS) ;

     // *** IMalloc methods **|
     STDMETHOD_ (void *, Alloc)        (THIS_ ULONG cb) ;
     STDMETHOD_ (void *, Realloc)      (THIS_ void *pv, ULONG cb) ;
     STDMETHOD_ (void,   Free)         (THIS_ void *pv) ;
     STDMETHOD_ (ULONG,  GetSize)      (THIS_ void *pv) ;
     STDMETHOD_ (int,    DidAlloc)     (THIS_ void *pv) ;
     STDMETHOD_ (void,   HeapMinimize) (THIS) ;

#ifndef CINTERFACE
public :
     DAlloc () ;
     ~DAlloc () ;
     BOOL Initialize () ;
private :
     ULONG RefCount ;
     HANDLE hHeap ;
#endif
     } ;

// Class Factory
#undef  INTERFACE
#define INTERFACE DClassFactory
```

```
DECLARE_INTERFACE_ (DClassFactory, IClassFactory)
     {
     // *** IUnknown methods **|
     STDMETHOD  (QueryInterface) (THIS_ REFIID riid, LPVOID FAR *ppv) ;
     STDMETHOD_ (ULONG, AddRef)  (THIS) ;
     STDMETHOD_ (ULONG, Release) (THIS) ;

     // *** IClassFactory methods **|
     STDMETHOD (CreateInstance) (THIS_ LPUNKNOWN pUnkOuter,
                                 REFIID riid, LPVOID FAR *ppvObject) ;
     STDMETHOD (LockServer)     (THIS_ BOOL fLock) ;

#ifndef CINTERFACE
public :
     DClassFactory () ;
     ~DClassFactory () ;
private :
     ULONG RefCount ;
#endif
     } ;
```

图 20-5 PUBMEM库

PUBMEM 有两个输出的入口点：DllGetClassObject 和 DllCanUnloadNow。由于 OLE 库希望 DLL 服务器输出完全相同的函数名，因此我们不能使用 EXPORT 关键字来定义这些函数，而是通过在 PUBMEM.MAK 文件中使用/EXPORT 连接选项来输出这些函数。DllGetClasObject 提供了输出类工厂的机制。当 OLE 库希望查询某个特定服务器是否拥有任何现存的连接，或者是否可以安全地终止服务器并从存储中删除它时，就调用 DllCanUnloadNow。后面将对二者加以简短讨论；先让我们来看一看登记表——使 OLE 组件公共化的一个关键因素。

20.4.1 登记表的作用

登记表是 Windows 95 和 Windows NT 中系统范围的固定状态信息的集中存放处。登记表是在 Windows 3.1 中引入的，用于公开 OLE 类细节、系统外壳的缺省扩展以及几个独特的 DDE 命令。在 Windows 95 和 Windows NT 中，登记表代替了过去位于.INI 文件中的状态，诸如当前安装的硬件、控制板选择以及用户偏好的应用软件设置。虽然这两个操作系统使用登记表的方式很不相同，但是 OLE 有关的登记表项在这两个操作系统中都是一样的。

登记表的结构是键(key)、子键(subkey)和值的一种层次结构。Windows 95 中定义的根键有 HKEY_CLASSES_ROOT、HKEY_CURRENT_USER、HKEY_LOCAL_MACHINE、HKEY_USERS、HKEY_CURRENT_CONFIG 和 HKEY_DYN_DATA。（Windows NT 只提供前四个。）有关 OLE 组件的数据位于 HKEY_MACHINE_ROOT 中（附带说一句，这一层次结构的真正位置是 HKEY_LOCAL_MACHINE\SOFTWARE\Classes，但是为了和 Windows 3.1 中的 OLE 应用程序兼容，OLE 类被赋予了它们自己的代理根节点）。

HKEY_CLASSES_ROOT 的根键由三种类型的项组成：文件扩展名、类名和系统项。

文件扩展名由一个句号(.)开始，将一个文件和一个复合文件服务器相关联（例如，“.vsd”对应 Visio Corparation 的 Visio）。类名为 OLE 组件类提供了可读的标识符，有两种用途。第一种是支持 OLE1.0 的复合文件服务器；这种用法是用诸如“Visio.Drawing.4”这样的类名来标识的。第二种是用于 OLE 动画。动画宏原语通过名字——在这种环境中，一个类名被称为一个“程序 ID”或者简称为“ProgID”——来创建 OLE 动画对象。例如，使用“Visio.Applicatio” ProgID，Visual Basic 程序可以通过动画方法和属性来创建并操纵 Visio 对象。

有三类系统登记表项：TypeLib、Interface 和 CLSID，其中的每一个都是它们自己的层次结构中的根。TypeLib 层次标识当前安装的类型库的位置。类型库是描述 OLE 组件内容的数据库。类型库广泛地用于动画支持，它被描述为所支持的所有接口的函数原型，并且还包含对帮助文件的引用，以便开发工具可以激活适当的帮助页来帮助宏程序员正确使用动画服务器。

Interface 层次包含一个根据接口 ID 排序的列表，其中列出了系统上安装的所有接口。它为接口提供可读的名字（Iunknown、IMalloc 等），并提供有关每个接口的细节（函数的个数和每个接口的基类）。

最后一个层次是 CLSID 层，它提供所有当前安装的（公共）OLE 组件的细节。一个 CLSID 是一个类标识符。与接口 ID（IID 和 REFIID 数据类型）一样，一个类 ID（CLSID 和 REFCLSID 数据类型）是一个 128 位（32 个十六进制位）的数字，提供一种机器可读的方式来唯一地标识一个组件类。与 IID 类型一样，CLSID 是 GUID 类型：

```
typedef struct _GUID
    {
    DWORD Data1 ;
    WORD  Data2 ;
    WORD  Data3 ;
    BYTE  Data4[8] ;
    } GUID ;
```

一个 CLSID 就像一个特定的 OLE 组件的电话号码。我们在前面对接口 ID（IID）的讨论中，将 IID 比作同一建筑物中部门间的电话分机号。从这一点看，CLSID 提供了从一个部门给另一个部门打电话的方式，用 OLE 的术语就是从一个组件到另一个组件。正如打电话给某公司工作的某人经常要求你首先拨通该公司的主机号，然后再请求一个分机号一样，和特定组件中的特定接口连接也要求你首先以 CLSID 来访问组件，然后再以 IID 来获得希望的接口。

继续我们的电话比喻，登记表就像 OLE 组件的一个全球电话号码簿。一个组件可以通过文件扩展名、类名或者类 ID 来引用。在这三项中，类 ID 项是最重要的，因为组件模块（.DLL 或者.EXE）的细节都存储在 CLSID 层次中。虽然你（或者你的安装软件）需要创建层次中的项，但是你不需要编程来访问它；而是调用 OLE 库，由这些接受类 ID 的库自己搜索登记表，并装入所希望的组件。

下面的登记表项使 PUBMEM 的组件成为公共可用的：

```
\HKEY_CLASSES_ROOT\
    CLSID\
```

```
{308D0430-1090-11cf-B92A-00AA006238F8}\
    InprocServer32 = C: \PETZOLD\CHAP20\PUBMEM.DLL
```

InprocServer32 意味着被引用的文件是一个 32 位的 DLL。其它的关键字还有一个 32 位 EXE 的 LocalServer32 、一个 16 位 DLL 的 InproServer 和一个 16 位 EXE 的 LocalServer 。为了公开这些其它类型服务器的存在，需要另外的登记表项。

有两种方式增加登记表项：调用登记表 API 或者使用某种调用登记表 API 的工具。如果要在登记表中增加一项而又不想写任何代码，可以运行登记表编辑器。Microsoft 的每种操作系统中都存在它的一个版本：在 Windows NT 3.51 中，运行 REGEDT32.EXE ；在 Windows 95 中，运行 REGEDIT.EXE 。本章前面已经提到过，与 OLE 有关的项出现在 HKEY_CLASSES_ROOT 层次结构下。

如果编程的话，你应该调用各种登记表编辑函数来修改登记表，这些函数具有 " Reg " 前缀。登记表层次中的每一个新层次都由一个登记表键句柄 HKEY 来表示。注册公共存储分配器 PUBMEM.DLL 涉及打开一个现存的登记表键（ RegOpernKeyEx ），创建两个新的键（ RegCreateKeyEx ），设置一个值（ RegSetValueEx ），然后关闭这三个打开的键（ RegCloseKey ）。假设组件服务器位于 C:\PETZOLD\CHAP20\PUBMEM.DL 中，那么登记 PUBMEM.DLL 的代码如图 20-6 中所示：

```
{
DWORD     dwDisp ;
HKEY      hkMain ;
HKEY      hkClass ;
HKEY      hkPath ;
LPCTSTR   lpClsid = "{308D0430-1090-11cf-B92A-00AA006238F8}" ;
LPCTSTR   lpPath  = "InprocServer32" ;
LPTSTR    lpValue = "C:\\PETZOLD\\CHAP20\\PUBMEM.DLL" ;

// Open "HKEY_CLASSES_ROOT\CLSID"
RegOpenKeyEx (HKEY_CLASSES_ROOT,
              "CLSID",
              0,
              KEY_ALL_ACCESS,
              &hkMain) ;

// Add \HKEY_CLASSES_ROOT\CLSID\{308...8F8}
RegCreateKeyEx (hkMain,
                lpClsid,
                0,
                "",
                REG_OPTION_NON_VOLATILE,
                KEY_ALL_ACCESS,
                NULL,
                &hkClass,
                &dwDisp) ;

if (dwDisp == REG_CREATED_NEW_KEY)
    {
    // Add \...\{308...8F8}\InprocServer32
    RegCreateKeyEx (hkClass,
                    lpPath,
```

```
                    0,
                    "",
                    REG_OPTION_NON_VOLATILE,
                    KEY_ALL_ACCESS,
                    NULL,
                    &hkPath,
                    &dwDisp) ;

        RegSetValueEx (hkPath,
                    "",
                    0,
                    REG_SZ,
                    (CONST BYTE *) lpValue,
                    lstrlen (lpValue) + 1) ;

        }

    RegCloseKey (hkPath) ;
    RegCloseKey (hkClass) ;
    RegCloseKey (hkMain) ;
    }
```

图 20-6 登记PUBMEM.DLL

登记表允许客户只要知道组件的 CLSID 就可以访问公共的组件服务器。让我们更仔细地看一下为这种数据类型产生唯一的值及其编码的技术。

20.4.2 产生和使用 CLSID 的技术

OLE 组件的唯一类 ID 对 OLE 组件的整体运作是很重要的，就如同唯一的电话号码对电话系统的整体运作很重要一样。如果没有特殊的安排，两个部门不能共享一个电话号码，对 OLE 组件也是如此：每个组件要求唯一的类 ID 以避免混淆。对网络 OLE 问题会更多，因为当更多的计算机——它们分别有自己的 OLE 组件——加入进来时，冗余类 ID 之间发生冲突的可能性增大了。

事实上，针对 OLE 组件的解决方案是来自网络计算领域。开放软件基金会（ OSF ）为它们的分布计算环境（ DCE ）标准创建了“全球唯一标识符”（ UUID ）。在 OLE 编程中， UUID 被称为 GUID，既用作组件类标识符（ CLSID ）又用作接口标识符（ IID ）。

运行 GUIDGEN 或者 UUIDGEN 工具可以产生一个唯一的 GUID，以用于标识组件类，或者标识你自己的客户接口。如 Kraig Brockschmidt 在 Inside OLE 中所描述的，这个程序使用网卡的唯一 IEEE 标识符、当前日期、当前时间和一个计数器（用于频繁的分配请求情况）的组合来产生唯一的值。结果是一个极不可能在不同开发者之间重复的数。

GUIDGEN 工具产生唯一的 GUID 并将这个值以几种形式之一的 CF_TEXE 格式放在剪贴板中，包括一种 MFC 特有的格式、一种原始 OLE （非 MFC ）格式和登记表格式的 GUID 串。下面是 PUBMEM 用来标识 IMalloc 分配器组件类的 GUIDGEN 输出：

```
// {308D0430-1090-11cf-B92A-00AA006238F8}
DEFIN_GUID (CLSID_ALLOCATOR, \
```

```
    0x308d0430, 0x1090, 0x11cf, 0xb9, \
    0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8) ;
```

注释行显示了登记表所需要的格式，其中在特定位置包含短划线并加上开始和结束花括号。符号化的标识符 CLSID_ALLOCATOR 被加在由 GUIDGEN 产生的结果中，以作为一个适用于 PUBMEM 组件的名字。

要正确地使用 DEFINE_GUID 宏需要小心一点，因为它有两种不同的定义。标准的定义把 GUID 名字作为一个 extern 值来引用。在你的一个并且仅有的一个源文件中，必须引用另一种定义，通过在 DEFINE_GUID 宏出现之前包含 INITGUID.H，很容易做到这一点：

```
// 重定义 DEFINE_GUID 来分配 GUID 存储
#include <initguid.h>

// 分配并且初始化 CLSID_ALLOCATOR
DEFINE_GUID  (CLSID_ALLOCATOR, \
    0x308d0430, 0x1090, 0x11cf, 0xb9, \
    0x2a, 0x0, 0xaa, 0x0, 0x62, 0x38, 0xf8) ;
```

建立一种 OLE 客户/服务器连接时，一个 CLSID 值从客户传递到 OLE 库，再由 OLE 库将这个值传递给服务器。客户使用类 ID 来请求对指定组件的访问。OLE 库使用类 ID 搜索系统登记表寻找与这个类 ID 相关联的程序或者 DLL 的位置。当最终找到该组件时，DLL 组件服务器中的元素——类工厂——对服务器支持所请求的 CLSID 进行核实。一旦得到证实，类工厂就自己来创建所请求的组件，并向客户提供第一个接口指针。

20.4.3 类工厂组件

类工厂是私有的 OLE 组件，支持两种接口：IUnknown 和 IClassFactory。它是私有组件，因为它没有类 ID，并且它的出现不会在系统登记表中公布。与 IMALLOC 中的分配器一样，客户通过私有入口点 DllGetClassObject 来访问类工厂。然而，与 IMALLOC 的分配器不一样的是，通常是 OLE 库自己在访问这个入口点并创建类工厂。

DllGetClassObject 函数的原型如下：

```
HRESULT
DllGetClassObject (REFCLSID rclsid,
    REFIID riid,
    LPVOID *ppv) ;
```

- rclsid 标识所希望的类（CLSID）的值。
- riid 标识所希望的接口（IID）的值，通常是 IID_ClassFactory。
- ppv 指向返回值将被写入的位置。

通过这个单一的输出入口点，DLL 服务器可以支持许多不同的类。DllGetClassObject 的第一个参数，rclsid，让服务器知道应该创建哪种特定的类。与 QueryInterface 函数一样，当 DllGetClassObject 可以提供所请求的接口时，它将一个接口指针——通常是一个 IClassFactory 接口指针——拷贝到由最后一个参数指定的位置。如果一个请求不能被满足，

那么最后一个参数将得到一个 NULL 值。

DllGetClassObject 的主要目的是检查是否支持一个特定的类（rclsid）和一个特定的接口（riid）。如果支持所请求的类，那么 PUBMEM 的 DllGetClassObject 创建一个类工厂。然后调用这个类工厂的 QueryInterface 函数——事实上，这一调用既是为了增加其引用计数也是为了检查是否支持所请求的接口。PUBMEM 中这个函数的版本如下所示：

```
HRESULT APIENTRY
DllGetClassObject (REFCLSID rclsid, REFIID riid, LPVOID *ppvObj)
     {
     // Initialize "but" pointer to known value
     *ppvObj = NULL ;

     if (rclsid != CLSID_ALLOCATOR)
          {
          return CLASS_E_CLASSNOTAVAILABLE ;
          }

     DClassFactory *pClassFactory = new DClassFactory () ;
     if (pClassFactory == NULL)
          {
          return E_OUTOFMEMORY ;
          }
     else
          {
          return pClassFactory->QueryInterface (riid, ppv) ;
          }
     }
```

就像 QueryInterface 一样，如果所请求的类或者所请求的类工厂接口不被支持，那么 DllGetClassObject 将返回的接口指针设置为 NULL。

IClassFactory 接口自身的定义如下所示：

```
#undef  INTERFACE
#define INTERFACE IClassFactory

DECLARE_INTERFACE_ (IClassFactory, IUnknown)
     {
     // *** IUnknown methods **|
     STDMETHOD (QueryInterface)   (THIS_
                                   REFIID riid,
                                   LPVOID FAR *ppvObj) PURE ;
     STDMETHOD_ (ULONG, AddRef)   (THIS) PURE ;
     STDMETHOD_ (ULONG, Release)  (THIS) PURE ;

     // *** IClassFactory methods **|
     STDMETHOD (CreateInstance)   (THIS_
                                   LPUNKNOWN pUnkOuter,
                                   REFIID riid,
                                   LPVOID FAR *ppvObject) PURE ;
     STDMETHOD (LockServer)       (THIS_ BOOL fLock) PURE ;
     } ;
```

与所有的 OLE 接口一样， IClassFactory 有一组通用的 IUnknown 成员函数，正是它们以及其它函数使 IClassFactory 成为一个独立而与众不同的组件。为了维持服务器对象和类工厂对象之间的这种区分，类工厂的 QueryInterface 不应该产生指向服务器对象的指针。反之亦然——服务器对象的 QueryInterface 不应该产生类工厂接口的指针。而应该让类工厂作为独立的组件，在需要时可以被创建并使用。

有两种特定于 IClassFactory 的服务：组件的创建和服务器的锁计数控制。

当客户希望创建组件的一个实例时，调用 CreateInstance 成员。然而，大多数客户不会直接调用这个函数，而是调用 OLE 库的辅助函数， CoCreateInstanct，来请求创建一个 OLE 组件。这个函数定义为：

```
STDAPI
CoCreateInstance (REFCLSID rclsid,              // 类标识符
                  LPUNKNOWN pUnkOuter,          // 外层 IUnknown
                  DWORD dwClsContext,           // 服务器环境
                  REFIID  riid,                 // 接口标识符
                  LPVOID *ppv) ;                // 返回值
```

- rclsid 标识要创建的组件的类 ID 。
- pUnkOuter 标识创建组合对象的外层 IUnknown 。
- dwClsContext 标识服务器组件的环境。可以选择包括下列值的组合：用于 DLL 服务器的 CLSCTX_INPROC_SERCER 、用于 DLL 句柄（进程外服务器的代理）的 CLSCTX_INPROC_HANDLER 和用于本地服务器（同一台机器上的不同进程）的 CLSCTX_LOCAL_SERVER 。从中可以看到，没有为网络服务器指定标志。
- riid 标识所请求的接口。
- 当组件创建成功时， ppv 指向返回值的位置，当不能使用所请求的接口创建组件时， ppv 为 NULL 。

IClassFactory 函数 LockServer 既增加又减少一个服务器的锁计数。它在单个服务器中结合了 IUnkown 的 AddRef 和 Release 成员函数为接口所做的工作。但有一点重要的区别，即：服务器永远不会像组件对象那样自己终止自己。这就引出了服务器实现中的一个重要问题：服务器生命周期的控制问题。

20.4.4 控制服务器的生命周期

服务器的生命周期问题对服务器的实现者来说，就如同组件的生命周期对组件的实现者一样重要。它在 PUBMEM 这样单组件的服务器中的重要性，并不亚于在多组件服务器中的重要性。控制这两种服务器生命周期的技术是相同的。我们的重点放在基于 DLL 的服务器的生命周期上。虽然对基于 EXE 的服务器来说，服务器生命周期的出发点是一样的，但是实现的细节却很不一样（有关细节，请参见 Brockschmidt 的“ Inside OLE ”）。

服务器 DLL 像其它任何 Windows DLL 那样装入存储——通过调用 LoadLibrary 。 OLE 库提供它们自己的 CoLoadLibrary 版本这一事实并不改变 DLL 之外的实体装入 DLL 这一事实（ CoLoadLibrary 被创建用来简化 OLE 向非 Windows 平台的移植)。后续 CoFreeLibrary

的调用将把 DLL 从存储中卸出。

DLL 服务器生命周期是一个问题，因为客户并不直接将服务器库装入存储，而是由 OLE 库来装入。因此应该由 OLE 库负责在正确的时间将服务器卸出。但是客户和服务器之间的大多数交互都忽略了 OLE 库，因此 OLE 库也无法知道什么时间是将 DLL 服务器从存储中删除的正确时间，除非是通过询问。

OLE 库周期性地调用服务器的 DllCanUnloadNow 入口点，以便确认是否到了卸出服务器的时间了。这个函数回答“是”(S_OK)或者“不”(S_FALSE):

```
HRESULT APIENTRY DllCanUnloadNow (void)
     {
     if (cObject > 0 || cLockCount > 0)
          {
          return S_FALSE ;
          }
     else
          {
          return S_OK ;
          }
     }
```

答案基于两个因素之一：是否有任何组件存在以及客户是否将服务器锁定在存储中。后者适用于，在没有组件存在时——为了避免重新将一个 DLL 装入存储的开销——客户希望服务器不要被终止的情况。

服务器上的锁计数是通过 IClassFactory 的 LockServer 成员来控制的。这个函数主要是对锁计数进行增减，使用锁计数比使用服务器来维护全局变量要好一些。下面是 PUBMEM 对 IClassFactory::LockServer 的实现：

```
STDMETHODIMP
DClassFactory::LockServer (BOOL fLock)
     {
     if (fLock)
          {
          ++cLockCount ;
          }
     else
          {
          --cLockCount ;
          }
     return NOERROR ;
     }
```

控制服务器生命周期的另一因素是组件的计数。在 PUBMEM 中，当所请求的 OLE 组件被成功创建时，这个计数在 DClassFactory::CreateInstance 中被加 1：

```
// Create memory allocation object
LPMALLOC pMalloc = CreateAllocator () ;

if (pMalloc == NULL)
     {
     return E_OUTOFMEMORY ;
```

```
        }
    else
        {
        // Fetch interface requested by caller
        HRESULT hr = pMalloc->QueryInterface (riid, ppvObject) ;

        // Decrement reference count produced by CreateAllocator
        pMalloc->Release () ;

        // Increment count of objects
        If (SUCCEEDED (hr))
            ++cObject ;

        return hr ;
        }
```

当一个组件清除自己时，这个计数减 1 。组件清除是在组件的引用计数变为 0 时由 IMalloc 的 Release 成员函数完成的:

```
STDMETHODIMP_ (ULONG)
DAlloc::Release (void)
     {
     if (0L != RefCount)
         return RefCount ;

     --cObject ;
     delete this ;
     return 0L ;
     }
```

有时，将私有组件转换为公共组件时，这是所需要进行的唯一改变。

现在是来看看为公共组件创建一个客户需要做些什么的时候了。下面的一个示例程序演示了这一点。

20.5 公共组件客户

这个公共组件客户 CALLPUB.EXE 是在私有组件客户 CALLER.EXE 基础上创建的。二者都调用 OLE 服务器组件，这些组件提供 IMalloc 接口来分配并释放容纳字符串的存储块。

虽然要将私有服务器转换为公共服务器（包括特殊的入口点和一个类工厂）需要进行相当的一些修改，但是只需要一个小小的改变，就可以将私有组件的客户转换为公共组件的客户。这里， CALLPUB 不再调用私有组件的创建函数，而是调用一个 OLE 库函数——CoCreateInstance ，该函数的唯一目的就是创建组件。

组成 CALLPUB 的源文件在图 20-7 中所示。

CALLPUB.MAK

```
#----------------------
```

```
# CALLPUB.MAK make file
#---------------------

callpub.exe : callpub.obj callpub.res pubmem.lib
     $(LINKER) $(GUIFLAGS) -OUT:callpub.exe callpub.obj callpub.res \
     pubmem.lib $(GUILIBS) uuid.lib ole32.lib

callpub.obj : callpub.c callpub.h pubmem.h
     $(CC) -DCINTERFACE $(CFLAGS) callpub.c

callpub.res : callpub.rc callpub.h
     $(RC) $(RCVARS) callpub.rc
```

CALLPUB.C

```
/*----------------------------------------------
   CALLPUB.C -- Call into public OLE component
                (c) Paul Yao, 1996
  ----------------------------------------------*/
#include <windows.h>
#include <initguid.h>
#include "pubmem.h"
#include "callpub.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szWndClass[] = "CallerWindow" ;
char szAppName[]  = "Calls Public Malloc" ;

//-------------------------------------------------------------------
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR lpszCmdLine, int cmdShow)
     {
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wc ;

     wc.cbSize          = sizeof (wc) ;
     wc.lpszClassName = szWndClass ;
     wc.hInstance       = hInstance ;
     wc.lpfnWndProc     = WndProc ;
     wc.hCursor         = LoadCursor (NULL, IDC_ARROW) ;
     wc.hIcon           = LoadIcon (NULL, IDI_APPLICATION) ;
     wc.lpszMenuName    = "MAIN" ;
     wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
     wc.style           = 0 ;
     wc.cbClsExtra      = 0 ;
     wc.cbWndExtra      = 0 ;
     wc.hIconSm         = LoadIcon (NULL, IDI_APPLICATION) ;

     RegisterClassEx (&wc) ;

     hwnd = CreateWindowEx (0L, szWndClass, szAppName,
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
```

```
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL) ;
     ShowWindow (hwnd, cmdShow) ;
     UpdateWindow (hwnd) ;

     // Connect to OLE libraries
     HRESULT hr = CoInitialize (NULL) ;
     if (FAILED (hr))
          {
          // Fail app initialization
          return FALSE ;
          }

     while (GetMessage (&msg, NULL, 0, 0))
          {
          TranslateMessage (&msg) ;
          DispatchMessage (&msg) ;
          }

     // Disconnect from OLE libraries
     CoUninitialize () ;

     return msg.wParam ;
     }

//-------------------------------------------------------------------
LRESULT CALLBACK
WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
     {
     static int       iCurLine = 0 ;
     static LPMALLOC  pMalloc = NULL ;
     static LPSTR     szLine[10] ;
     static RECT      rHit[10] ;

     switch (iMsg)
          {
          case WM_CREATE :
               // Initialize data pointer array
               ZeroMemory (szLine, sizeof (szLine)) ;
               return 0 ;

          case WM_COMMAND :
               switch (LOWORD (wParam))
                    {
                    case IDM_CREATE :
                         {
                         HRESULT hr =
                              CoCreateInstance (CLSID_ALLOCATOR,
                                                NULL,
                                                CLSCTX_INPROC_SERVER,
                                                IID_IMalloc,
                                                (void **) &pMalloc) ;

                         if (FAILED (hr))
                              {
```

```
                MessageBox (hwnd, "Error: No allocator",
                            szAppName, MB_OK) ;
                return 0 ;
                }

           InvalidateRect (hwnd, NULL, TRUE) ;
           return 0 ;
           }

      case IDM_DESTROY :
           {
           int i ;

           // Mark allocated blocks as invalid
           for (i = 0 ; i < 10 ; i++)
                {
                if ((szLine[i] != NULL) &&
                    (pMalloc->lpVtbl->DidAlloc (pMalloc,
                                                szLine[i])))
                     {
                     szLine[i] = NULL ;
                     }
                }
           // Disconnect from & free allocator
           pMalloc->lpVtbl->Release (pMalloc) ;
           pMalloc = NULL ;

           InvalidateRect (hwnd, NULL, TRUE) ;
           return 0 ;
           }

      case IDM_IUNKNOWN :
           {
           LPUNKNOWN pUnk ;
           HRESULT hr =
                pMalloc->lpVtbl->QueryInterface (pMalloc,
                     IID_IUnknown,
                     (void **) &pUnk) ;
           if (SUCCEEDED (hr))
                {
                pUnk->lpVtbl->Release (pUnk) ;
                MessageBox (hwnd, "IUnknown supported",
                            szAppName, MB_OK) ;
                }
           else
                {
                MessageBox (hwnd, "IUnknown not supported",
                            szAppName, MB_OK) ;
                }
           return 0 ;
           }

      case IDM_IMALLOC :
           {
           LPUNKNOWN pUnk ;
```

```
        HRESULT hr =
             pMalloc->lpVtbl->QueryInterface (pMalloc,
                  IID_IMalloc,
                  (void **) &pUnk) ;

        if (SUCCEEDED (hr))
             {
             pUnk->lpVtbl->Release (pUnk) ;
             MessageBox (hwnd, "IMalloc supported",
                         szAppName, MB_OK) ;
             }
        else
             {
             MessageBox (hwnd, "IMalloc not supported",
                         szAppName, MB_OK) ;
             }
        return 0 ;
        }

   case IDM_IMARSHAL :
        {
        LPUNKNOWN pUnk ;
        HRESULT hr =
             pMalloc->lpVtbl->QueryInterface (pMalloc,
                  IID_IMarshal,
                  (void **) &pUnk) ;
        if (SUCCEEDED (hr))
             {
             pUnk->lpVtbl->Release (pUnk) ;
             MessageBox (hwnd, "IMarshal supported",
                         szAppName, MB_OK) ;
             }
        else
             {
             MessageBox (hwnd, "IMarshal not supported",
                         szAppName, MB_OK) ;
             }
        return 0 ;
        }

   case IDM_ALLOCATE_CUSTOM :
        if (szLine[iCurLine] != NULL)
             {
             MessageBox (hwnd, "Error: Free First",
                         szAppName, MB_OK) ;
             return 0 ;
             }

        // Allocate from IMalloc interface
        szLine[iCurLine] =
             (char *) pMalloc->lpVtbl->Alloc (pMalloc, 100) ;
        lstrcpy (szLine[iCurLine], "*IMalloc memory*") ;

        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;
```

```
                    case IDM_ALLOCATE_DEFAULT :
                         if (szLine[iCurLine] != NULL)
                              {
                              MessageBox (hwnd, "Error: Free First",
                                          szAppName, MB_OK) ;
                              return 0 ;
                              }

                         // Allocate from default heap
                         szLine[iCurLine] = (char *) malloc (100) ;
                         lstrcpy (szLine[iCurLine], "-Malloc memory-") ;

                         InvalidateRect (hwnd, NULL, TRUE) ;
                         return 0 ;

                    case IDM_FREE :
                         if (szLine[iCurLine] == NULL)
                              {
                              MessageBox (hwnd, "Error: Nothing to free",
                                          szAppName, MB_OK) ;
                              return 0 ;
                              }

                         if (pMalloc == NULL)
                              {
                              goto FreeMalloc ;
                              }

                         // Free allocated object
                         if (pMalloc->lpVtbl->DidAlloc (pMalloc,
                                                szLine[iCurLine]))
                              {
                              pMalloc->lpVtbl->Free (pMalloc,
                                                 szLine[iCurLine]) ;
                              }
                         else
                              {
               FreeMalloc:
                              free (szLine[iCurLine]) ;
                              }

                         szLine[iCurLine] = NULL ;

                         InvalidateRect (hwnd, NULL, TRUE) ;
                         return 0 ;
                    }

          case WM_DESTROY :
               // Disconnect from & free allocator
               if (pMalloc)
                    {
                    pMalloc->lpVtbl->Release (pMalloc) ;
                    pMalloc = NULL ;
                    }
```

```
          PostQuitMessage (0) ;  // Handle application shutdown
          return 0 ;

     case WM_INITMENU :
          {
          HMENU hMenu = (HMENU) wParam ;
          if (pMalloc)
               {
               EnableMenuItem (hMenu, IDM_CREATE,          MF_GRAYED) ;
               EnableMenuItem (hMenu, IDM_DESTROY,         MF_ENABLED) ;
               EnableMenuItem (hMenu, IDM_ALLOCATE_CUSTOM, MF_ENABLED) ;
               EnableMenuItem (hMenu, IDM_IUNKNOWN,        MF_ENABLED) ;
               EnableMenuItem (hMenu, IDM_IMALLOC,         MF_ENABLED) ;
               EnableMenuItem (hMenu, IDM_IMARSHAL,        MF_ENABLED) ;
               }
          else
               {
               EnableMenuItem(hMenu, IDM_CREATE,          MF_ENABLED) ;
               EnableMenuItem(hMenu, IDM_DESTROY,         MF_GRAYED) ;
               EnableMenuItem(hMenu, IDM_ALLOCATE_CUSTOM, MF_GRAYED) ;
               EnableMenuItem(hMenu, IDM_IUNKNOWN,        MF_GRAYED) ;
               EnableMenuItem(hMenu, IDM_IMALLOC,         MF_GRAYED) ;
               EnableMenuItem(hMenu, IDM_IMARSHAL,        MF_GRAYED) ;
               }
          return 0 ;
          }

     case WM_LBUTTONDOWN :
          {
          int i ;
          int x = LOWORD (lParam) ;
          int y = HIWORD (lParam) ;
          POINT pt = { x, y } ;

          for (i = 0 ; i < 10 ; i++)
               {
               if (PtInRect (&rHit[i], pt))
                    {
                    if (iCurLine != i)  // Minimize screen blink
                         {
                         InvalidateRect (hwnd, &rHit[iCurLine], TRUE) ;
                         InvalidateRect (hwnd, &rHit[i], TRUE) ;
                         iCurLine = i ;
                         }
                    break ;
                    }
               }
          return 0 ;
          }

     case WM_PAINT :
          {
          char          szBuff[10] ;
          COLORREF      crText, crBack ;
```

```
HDC          hdc ;
int          cc ;
int          i ;
int          XCount, XText, Y ;
int          cyLineHeight ;
PAINTSTRUCT  ps ;
RECT         rOpaque ;
TEXTMETRIC   tm ;

hdc = BeginPaint (hwnd, &ps) ;

// Fetch line height
GetTextMetrics (ps.hdc, &tm) ;
cyLineHeight = tm.tmHeight + tm.tmExternalLeading ;

// Fetch current text colors
crText = GetTextColor (ps.hdc) ;
crBack = GetBkColor (ps.hdc) ;

XCount = tm.tmAveCharWidth * 3 ;
XText  = XCount + tm.tmAveCharWidth * 7 ;
Y      = tm.tmHeight ;

for (i = 0 ; i < 10 ; i++, Y += cyLineHeight)
     {
     // Set colors to highlight current line
     if (i == iCurLine)
          {
          SetTextColor (ps.hdc, crBack) ;
          SetBkColor (ps.hdc, crText) ;

          SetRect (&rOpaque, 0, Y, 9999, Y + cyLineHeight) ;
          ExtTextOut(ps.hdc, 0, 0, ETO_OPAQUE, &rOpaque,
                     NULL, 0, NULL ) ;
          }
     else
          {
          SetTextColor (ps.hdc, crText) ;
          SetBkColor (ps.hdc, crBack) ;
          }

     // Display line count
     cc = wsprintf (szBuff, "Line %d", i) ;
     TextOut (ps.hdc, XCount, Y, szBuff, cc) ;

     // Display text if a string has been defined
     if (szLine[i] != NULL)
          {
          cc = lstrlen (szLine[i]) ;
          TextOut (ps.hdc, XText, Y, szLine[i], cc) ;
          }

     // Calculate hit test rectangle
     SetRect (&rHit[i], 0, Y, 9999, Y + cyLineHeight) ;
     }
```

```
            EndPaint (hwnd, &ps) ;
            return 0 ;
            }

        default :
            return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
        }
    }
```

CALLPUB.H

```
#define IDM_CREATE              1
#define IDM_DESTROY             2
#define IDM_IUNKNOWN            3
#define IDM_IMALLOC             4
#define IDM_IMARSHAL            5
#define IDM_ALLOCATE_CUSTOM     6
#define IDM_ALLOCATE_DEFAULT    7
#define IDM_FREE                8
#define IDM_CHECK               9
```

CALLPUB.RC

```
#include "callpub.h"

MAIN MENU
     {
     POPUP "&Allocator"
          {
          MENUITEM "&Create",                      IDM_CREATE
          MENUITEM "&Destroy",                     IDM_DESTROY
          MENUITEM SEPARATOR
          MENUITEM "QueryInterface IID_IUnknown", IDM_IUNKNOWN
          MENUITEM "QueryInterface IID_IMalloc",  IDM_IMALLOC
          MENUITEM "QueryInterface IID_IMarshal", IDM_IMARSHAL
          }

     POPUP "&Memory"
          {
          MENUITEM "&Allocate (IMalloc)", IDM_ALLOCATE_CUSTOM
          MENUITEM "&Allocate (malloc)",  IDM_ALLOCATE_DEFAULT
          MENUITEM "&Free",               IDM_FREE
          }
     }
```

图 20-7　CALLPUB程序

由于它调用了 OLE 库函数，因此 CALLPUB 在启动时从 WinMain 中对 OLE 库进行初始化：

```
// 与 OLE 库连接
HRESULT hr = CoInitialize (NULL) ;
if (FAILED (hr))
    {
    // 失败的应用初始化
    return FALSE ;
    }
```

并在终止之前切断和 OLE 的连接:

```
// 切断与 OLE 库的连接
CoUnintialize ( ) ;
```

这两个调用都是必要的，因为 CALLPUB 是通过调用 CoCreateInstance，从 PUBMEM 组件服务器中获得它的 IMalloc 接口指针的:

```
HRESULT hr =
    CoCreateInstance (CLSID_ALLOCATOR,
                      NULL,
                      CLSCTX_INPROC_SERVER,
                      IID_IMalloc,
                      (void**) &pMalloc) ;
if (FAILED (hr))
    {
    MessageBox (hwnd, "Error: No allocator",
                szAppName, MB_OK) ;
    }
```

这个 OLE 库函数搜索登记表，寻找哪个组件可以提供 CLSID_ALLOCATOR 类，CLSID_ALLOCATOR 是定义在 PUBMEM.H 中的一个符号。为了强迫这个符号分配存储而不是简单地引用一个外部的值，PUBMEM 引用了下面的头文件:

```
#include <initguid.h>
#include "pubmem.h"
```

到编写这本书时，为可运行服务器定义了下面三种环境（CoCreateInstance 中的 dwClsContext 参数）。PUBMEM 请求的 CLSCTX_INPROC_SERVER 是一个独立的 DLL 服务器。另一种类型的 DLL 服务器 CLSCTX_INPROC_HANDLER 是进程外服务器的本地句柄，它担任进程内代理的角色（更快、开销更小），直到确实需要进程外的服务器时为止。第三种环境 CLSCTX_LOCAL_SERVER 是在同一台机器上运行的一个独立进程。

私有的 CreateAllocator 函数总是返回指向 IMalloc 接口的指针，但 CoCreateInstance 不一样，它让调用者指定要返回的接口的类型。PUBMEM 请求 IID_IMalloc，这是 IMalloc 的接口 IID。但是它也可以请求 IID_IUnknown，然后对 IMalloc 指针执行一个 QueryInterface，最终达到同样的结果。当然，如果两个接口指针由同一个客户所拥有，那么最终将需要对 Release 的两次调用来正确地减小引用计数。

20.6 最后的话

OLE 组件对象模型（COM）的基础介绍应该能够为你提供足够的背景以便开始使用 OLE 技术。不管是创建复合文件服务器（容器），还是创建动画对象（控件）、或者 OLE 控件容器、或者 OLE 控件，这都依赖于本章所描述的 COM 接口。Windows 95 的外壳服务只有通过 COM 接口才可用。即使你是基于一个类库，诸如 Microsoft 的 MFC 或者 Borland

的 OWL，但是只要你稍微深入一些，将会发现 COM 出现在你眼前。

在 Windows 95 开发的初期，Microsoft 就传递给软件开发人员一条始终如一的消息：未来是属于 Win32 和 OLE 的。随着 Windows 95 的到来和 OLE 技术美好前景的展现，未来已经来到了我们的面前。

现在，你拥有了创建明天的优秀 Windows 应用程序所需要的知识。

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):
{
"filename": "44CKV2luZG93czk156iL5bqP6K6+6K6h44CLXzEwMjc2OTc0LnppcA==",
"filename_decoded": "\u300aWindows95\u7a0b\u5e8f\u8bbe\u8ba1\u300b_10276974.zip",
"filesize": 69955825,
"md5": "4af859f792984827a41e67fa58cd9e81",
"header_md5": "d84002731243b53217be3183d44643c7",
"sha1": "84ee58949518573a88becfaf5af8b059bce2e26e",
"sha256": "6cd197139ea705f785d2f088a1efb74e8e3e877705180f9ad3557d036c494a46",
"crc32": 1403562975,
"zip_password": "",
"uncompressed_size": 74305060,
"pdg_dir_name": "\u00ed\u2562Windows95\u2502\u2560\u2568\u2265\u2554\u03a6\u255d\u255e\u00ed\u2556_10276974",
"pdg_main_pages_found": 875,
"pdg_main_pages_max": 875,
"total_pages": 892,
"total_pixels": 6265228952,
"pdf_generation_missing_pages": false
}